FORM PTO-1390 (REV. 5-93)	U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE	ATTORNEY'S DOCKET NUMBER 2885/56
TRANSMITTAL LETTER TO THE UNITED STATES DESIGNATED/ELECTED OFFICE (DO/EO/US) CONCERNING A FILING UNDER 35 U.S.C. 371		U.S APPLICATION NO. (If known, see 37 CFR 1.5) **10/009649**

INTERNATIONAL APPLICATION NO. PCT/DE00/01869	INTERNATIONAL FILING DATE (13.06.00) 13 June 2000	PRIORITY DATE(S) CLAIMED (10.06.99) (09.01.00) (12.04.00) 10 June 1999 09 January 2000 12 April 2000

TITLE OF INVENTION
PROGRAMMING CONCEPTS

APPLICANT(S) FOR DO/EO/US

VORBACH, Martin; NÜCKEL, Armin

Applicant(s) herewith submit to the United States Designated/Elected Office (DO/EO/US) the following items and other information

1. ☒ This is a **FIRST** submission of items concerning a filing under 35 U.S.C. 371.

2. ☐ This is a **SECOND** or **SUBSEQUENT** submission of items concerning a filing under 35 U.S.C. 371.

3. ☐ This is an express request to begin national examination procedures (35 U.S.C 371(f)) immediately rather than delay examination until the expiration of the applicable time limit set in 35 U.S.C. 371(b) and PCT Articles 22 and 39(1).

4. ☒ A proper Demand for International Preliminary Examination was made by the 19th month from the earliest claimed priority date.

5. ☒ A copy of the International Application as filed (35 U.S.C. 371(c)(2))

   a. ☐ is transmitted herewith (required only if not transmitted by the International Bureau).

   b. ☒ has been transmitted by the International Bureau.

   c. ☐ is not required, as the application was filed in the United States Receiving Office (RO/US)

6. ☒ A translation of the International Application into English (35 U S C. 371(c)(2)).

7. ☒ Amendments to the claims of the International Application under PCT Article 19 (35 U.S.C. 371(c)(3))

   a. ☐ are transmitted herewith (required only if not transmitted by the International Bureau)

   b. ☐ have been transmitted by the International Bureau.

   c. ☐ have not been made; however, the time limit for making such amendments has NOT expired.

   d. ☒ have not been made and will not be made.

8. ☐ A translation of the amendments to the claims under PCT Article 19 (35 U.S.C 371(c)(3)).

9. ☒ An oath or declaration of the inventor(s) (35 U S.C. 371(c)(4)) (unsigned).

10. ☐ A translation of the annexes to the International Preliminary Examination Report under PCT Article 36 (35 U.S.C. 371(c)(5)).

Items 11. to 16. below concern other document(s) or information included:

11. ☒ An Information Disclosure Statement under 37 CFR 1.97 and 1 98.

12. ☐ An assignment document for recording   A separate cover sheet in compliance with 37 CFR 3.28 and 3 31 is included.

13. ☒ A **FIRST** preliminary amendment

   ☐ A **SECOND** or **SUBSEQUENT** preliminary amendment.

14. ☐ A substitute specification and a marked up version thereof.

15. ☐ A change of power of attorney and/or address letter.

16. ☒ Other items or information: copies of International Search Report (translated) and Form PCT/RO/101.

Express Mail No. EV0037335200US

U.S. APPLICATION NO if known, see 37 C.F.R. 1.5 **10/009649**	INTERNATIONAL APPLICATION NO **PCT/DE00/01869**	ATTORNEY'S DOCKET NUMBER **2885/56**

17. ☒ The following fees are submitted:

Basic National Fee (37 CFR 1.492(a)(1)-(5)):
Search Report has been prepared by the EPO or JPO $890.00

International preliminary examination fee paid to USPTO (37 CFR 1 482) . . . $710.00

No international preliminary examination fee paid to USPTO (37 CFR 1.482) but international search fee paid to USPTO (37 CFR 1.445(a)(2)) $740.00

Neither international preliminary examination fee (37 CFR 1.482) nor international search fee (37 CFR 1 445(a)(2)) paid to USPTO $1,040.00

International preliminary examination fee paid to USPTO (37 CFR 1.482) and all claims satisfied provisions of PCT Article 33(2)-(4) $100.00

	CALCULATIONS	PTO USE ONLY
ENTER APPROPRIATE BASIC FEE AMOUNT =	$ 890	
Surcharge of $130.00 for furnishing the oath or declaration later than ☐ 20 ☐ 30 months from the earliest claimed priority date (37 CFR 1.492(e)).	$	

Claims	Number Filed	Number Extra	Rate		
Total Claims	1 - 20 =	0	X $18.00	$ .00	
Independent Claims	1 - 3 =	0	X $84.00	$ 00	
Multiple dependent claim(s) (if applicable)			+ $280.00	$	
TOTAL OF ABOVE CALCULATIONS =				$890.00	
Reduction by 1/2 for filing by small entity, if applicable. Verified Small Entity statement must also be filed. (Note 37 CFR 1 9, 1 27, 1.28).				$445.00	
SUBTOTAL =				$445.00	
Processing fee of $130.00 for furnishing the English translation later than ☐ 20 ☐ 30 months from the earliest claimed priority date (37 CFR 1.492(f)). +				$	
TOTAL NATIONAL FEE =				$445.00	
Fee for recording the enclosed assignment (37 CFR 1.21(h)). The assignment must be accompanied by an appropriate cover sheet (37 CFR 3.28, 3.31). $40.00 per property +				$	
TOTAL FEES ENCLOSED =				$445 00	
				Amount to be: refunded	$
				charged	$

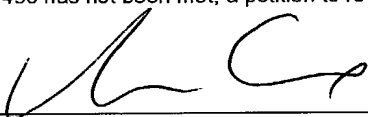a. ☐ A check in the amount of $_____ to cover the above fees is enclosed

b. ☒ Please charge my Deposit Account No. __11-0600___ in the amount of $445.00 to cover the above fees. A duplicate copy of this sheet is enclosed.

c. ☒ The Commissioner is hereby authorized to charge any additional fees which may be required, or credit any overpayment to Deposit Account No. ___11-0600___. A duplicate copy of this sheet is enclosed.

NOTE: Where an appropriate time limit under 37 CFR 1.494 or 1 495 has not been met, a petition to revive (37 CFR 1.137(a) or (b)) must be filed and granted to restore the application to pending status.

SEND ALL CORRESPONDENCE TO:

_____
SIGNATURE

Kenyon & Kenyon
One Broadway
New York, New York 10004
CUSTOMER NO. 26646

Michelle M Carniaux (Reg. No. 36,098)
NAME

/o Dec 2001
DATE

NY01 429851 v 1

[2885/56]

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s)   :   Martin VORBACH, et al.

Serial No.   :   To be assigned

Filed   :   Herewith

For   :   PROGRAMMING CONCEPTS

Examiner   :   To Be Assigned

Art Unit   :   To Be Assigned

Assistant Commissioner
   for Patents
Washington D.C.   20231

PRELIMINARY AMENDMENT

SIR:

        Kindly amend the above-identified application before examination, as set forth below.

In the Claims
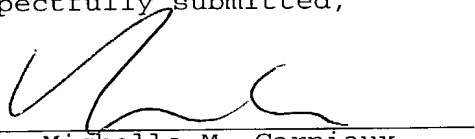
Please cancel claims 2-178 without prejudice.

Remarks

    Applicants assert that the present invention is new, non-obvious, and useful. Prompt consideration and allowance of the present application are earnestly solicited.

                                Respectfully submitted,

Dated: _10 Dec 2001_   By: _____
                                    Michelle M. Carniaux
                                    Reg. No. 36,098

                                KENYON & KENYON
                                One Broadway
                                New York, NY   10004
                                (212) 425-7200

(54) Title: SEQUENCE PARTITIONING IN CELL STRUCTURES

(54) Bezeichnung: SEQUENZ-PARTITIONIERUNG AUF ZELLSTRUKTUREN
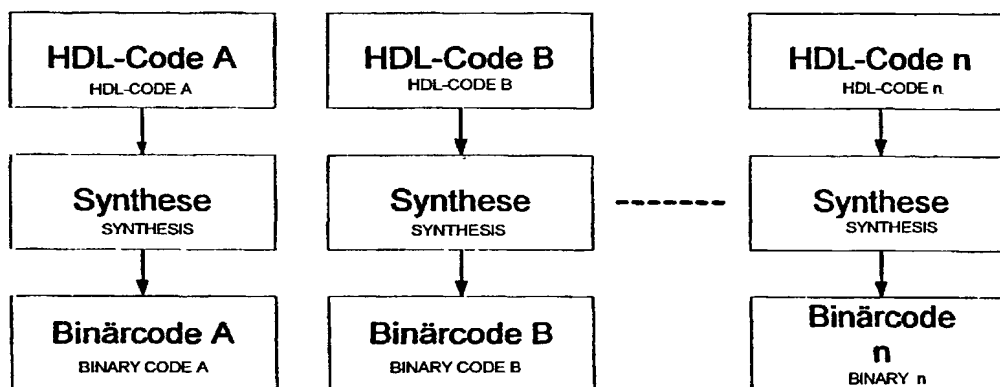
Stand der Technik
STATE OF THE ART

(57) **Abstract:** The invention relates to cell structures which can be variably arranged in relation to each other. The invention notably specifies how and with which units a sequence is to be partitioned for this purpose and with such cell structures.

(57) **Zusammenfassung:** Die vorliegende Erfindung befasst sich mit Zellstrukturen, bei denen eine wechselnde Anordnung zueinander möglich ist. Es wird angegeben, wie und mit welchen Einheiten hierbei und hierfür eine Sequenz zu partitionieren ist.

PROGRAMMING CONCEPTS

Object of the Invention and Areas of Application

The present invention extends to the domain of programmable
arithmetic and/or logic modules (VPUs), which can be

5   reprogrammed during operation in particular, having a
plurality of arithmetic and/or logic units whose
interconnection can also be programmed and reprogrammed during
operation. Such logical modules are available from several
manufacturers under the generic name of FPGA (Field-

10   Programmable Gate Arrays). Furthermore, several patents have
been published, which disclose special arithmetic modules
having automatic data synchronization and improved arithmetic
data processing.

15   All the above-described modules have a two-dimensional or
multidimensional arrangement of logical and/or arithmetic
units (Processing Array Elements -- PAEs) which can be
interconnected via bus systems.

20   Characteristic for the modules according to the present
invention is that they either have the units listed below or
these units are programmed or added (including externally) for
the use according to the present invention:
1.   at least one unit (CT) for loading the configuration

25   data;
2.   PAEs;
3.   at least one interface (IOAG) for one or more memory(ies)
      and/or peripheral device(s).

30   The object of the present invention is to provide a
programming method which allows the above-described modules to
be efficiently programmed in the known high-level programming
languages, making automatic, full, and efficient use of the
parallelism of the above-described modules obtained by the

NY01 419288   **LITERAL TRANSLATION**

plurality of units to the maximum possible degree.

Background Information

5   Modules of the type mentioned in the preamble are mostly programmed using popular data flow languages. This creates two basic problems:

1.   Programming in data flow languages requires getting used to by the programmer; multilevel sequential tasks can only be

10   described in a complex manner;

2.   Large applications and sequential descriptions can be mapped to the desired target technology (synthesized) with the existing translation programs (synthesis tools) only to a certain extent.

15

In general, applications are partitioned into multiple subapplications, which are then synthesized to the target technology individually (Fig. 1). Each of the individual binary codes is then loaded onto one module. The basic

20   prerequisite of the present invention is the method described in German Patent 44 16 881, which makes it possible to use a plurality of partitioned subapplications within a single module by analyzing the time dependence, sequentially requesting the required subapplications from a higher-level

25   load unit via control signals, whereupon the load unit loads the supplications onto the module.

Existing synthesis tools are capable of mapping program loops onto modules only to a certain extent (Fig. 2 0201). **FOR** loops

30   (0202) are often only supported as primitive loops by fully rolling out the loop onto the resources of the target module.

Contrary to FOR loops, **WHILE** loops (0203) have no constant abort value. Instead, it is evaluated using a condition,

35   whenever interrupt takes place. Therefore, normally (when the condition is not constant), at the time of the synthesis it is

not known when the loop is aborted. Due to their dynamic
behavior, these synthesis tools cannot map these loops onto
the hardware, i.e., transfer them to a target module, in a
fixed manner.

5

Recursions basically cannot be mapped onto hardware according
to the related art using synthesis tools if the recursion
depth is not known, i.e., constant, at the time of the
synthesis. When recursion is used, new resources are allocated

10   with each new recursion level. This would mean that new
hardware has to be made available with each recursion level,
which, however, is dynamically impossible.

Even simple basic structures can only be mapped by synthesis

15   tools when the target module is sufficiently large, i.e.,
offers sufficient resources.

Simple time dependencies (0301) are not partitioned into
multiple subapplications by today's synthesis tools and can

20   therefore only be transferred onto a target module as a whole.

Conditional executions (0302) and loops over conditions (0303)
can also only be mapped if sufficient resources exist on the
target module.

25

Method According to the Present Invention

The method described in German Patent 44 16 881 allows
conditions to be recognized within the hardware structures of

30   the above-mentioned modules at runtime and makes it possible
to dynamically respond to such conditions so that the function
of the hardware is modified according to the condition
received, which is basically accomplished by configuring a new
structure.

35

One essential step in the method according to the present

invention is the partitioning of graphs (applications) into time-independent subgraph s (subapplications).

The term "time independence" is defined so that the data which are transmitted between two subapplications are separated by a memory of any design (i.e., including simple register). This is possible, in particular, at the points of a graph where there is a clear interface with a limited and minimum amount of signals between the two subgraphs.

Furthermore, points in the graph having the following features are suitable in particular:

1. There are few signals or variables between the nodes;
2. A small amount of data is transmitted via the signals or variables;
3. There is no feedback, i.e., no signals or variables are transmitted in the direction opposite to the others.

In the case of large graphs, time independence can be achieved by introducing specific, clearly defined interfaces that are as simple as possible to store data in a buffer (see S_n in Fig. 4).

Loops often have a strong time independence with respect to the rest of the algorithm, since they work over a long period on a certain amount of variables that are (mostly) local in the loop and require a transfer of operands or of the result only when entering or leaving the loop.

With time independence it is achieved that after a subapplication has been completely executed, the subsequent subapplication can be loaded without any further dependencies or influences occurring. When the data is stored in the above-named memory, a status signal (trigger, see PACT08) can be generated, which requests the higher-level load unit to load the next subapplication. When simple registers are used as

LITERAL TRANSLATION

memories, the trigger can always be generated when data is written into the register. When memories are used, in particular memories operating by the FIFO principle, triggers are generated depending on multiple conditions. For example,

5   the following conditions, individually or in combination, can generate a trigger:
- Result memory full
- Operand memory empty
- No new operands
10 - Any condition within the subapplication, generated, e.g., by
    • comparators (equal, greater, etc.)
    • counters (overrun)
    • adders (overrun)

15 In the following, a subapplication will also be referred to as a module in order to improve understandability from the point of view of conventional programming. For the same reason, signals will also be called variables in the following. These variables differ from conventional variables in one important
20 aspect: a status signal (Ready) which shows whether this variable has a legal value is assigned to each variable. If a signal has a legal (calculated) value, the status signal is Ready; if the signal has no legal value (calculation not yet completed), the status signal is Not_Ready. The principle is
25 described in detail in Patent Application P196 51 075.9.

In summary, the following functions can be assigned to the triggers:

30 1. Control of data processing as the status of individual PAEs;
2. Control of reconfiguration of PAEs (time sequence of the subapplications).

35 In particular, the abort criteria of loops (WHILE) and recursions, as well as conditional jumps in subapplications, are implemented by triggers.

In case 1, the triggers are exchanged between PAEs; in case 2, the triggers are transmitted by the PAEs to the CT. What is essential to the present invention is that the transition between case 1 and case 2 basically depends on the number of
5   subapplications running at the time in the matrix of PAEs. In other words, triggers are sent to the subapplications currently being executed on the PAEs. If a subapplication is not configured, the triggers are sent to the CT. It is important that if this subapplication were also configured,
10   the respective triggers would be sent directly to the respective PAEs.

This results in automatic scaling of the computing performance with increasing PAE size, i.e., with cascading of a plurality
15   of PAE matrices. No more reconfiguration time is needed, but the triggers are sent directly to the PAEs which are now already configured.

Wave reconfiguration

20

A plurality of modules can be overlapped using appropriate hardware architecture (see Figs. 10/11). This means that a plurality of modules are pre-configured in the PAEs at the same time and switching between configurations can be
25   performed with minimum expenditure in time, so that only precisely one configuration is ever activated out of a plurality of configurations for each PAE.

It is important that in a number of PAEs into which a module A
30   and a module B is preconfigured, one part of this number can be activated using a part of A and another part of this number can be activated at the same time using a part of B. The separation of the two parts is given exactly by the PAE in which the switch-over state between A and B occurs. This means
35   that, from a certain point in time B is activated in all PAEs for which A was activated for execution prior to this time,

and in all other PAEs A is still activated after this time. With increasing time, B is activated in more and more PAEs.

Switch-over takes place on the basis of specific data, states,
5   which result from the computation of the data or on the basis of any other events which are generated externally or by the CT, for example.

As a result, after a data packet has been processed, switch-
10   over to another configuration can take place. At the same time/alternatively, a signal (RECONFIG-TRIGGER) can be sent to the CT, which causes new configurations to be pre-loaded by the CT. Pre-loading can take place onto other PAEs, which are dependent on or independent of the current data processing. By
15   isolating the active configuration from the configurations which are now available for reconfiguration (see Figs. 10/11), new configurations can be loaded even into PAEs that are currently operating (active), in particular also the PAE which generated the RECONFIG-TRIGGER. This allows a configuration to
20   overlap with the data processing.

Figure 13 shows the basic principle of wave reconfiguration (WRC). It is based on a row of PAEs (PAE1 - PAE9), through which the data runs as through a pipeline. It should be
25   specifically pointed out that. WRC is not limited to pipelines and the interconnection and grouping of PAEs may assume any desired form. The illustration was selected in order to show a simple example for easier understanding.

30   In Fig. 13a, a data packet runs in PAE1. The PAE has four possible configurations (A, F, H, C), which can be selected using appropriate hardware (see Figs. 10/11). Configuration F is activated in PAE1 for the current data packet (shaded area).
35

In the next cycle, the data packet runs to PAE2 and a new data packet appears in PAE1. F is also active in PAE2. Together with the data packet, an event (I1) appears in PAE1. The event

occurs whenever the PAE receives any external event (e.g., a status flag or a trigger) or it is generated within the PAE by the computation performed.

5   In Fig. 13c, configuration H is activated in PAE1 because of the event (I1); at the same time, a new event (I2) appears, which causes configuration A to be activated in the following cycle (Fig. 13d).

10   In Fig. 13e, (I3) is received at PAE1, which causes F to be overwritten by G (Fig. 13f). G is activated with the receipt of (I4) (Fig. 13g). (I5) causes K to be loaded instead of C (Fig. 13h, i), and (I6) loads and starts F instead of H (Fig. 13j).

15
       Figs. 13g*) to 13j*) show that when running a wave reconfiguration, not all PAEs need to operate according to the same pattern. The way a PAE is configured by a wave configuration depends mainly on its own configuration. It
20   should be mentioned here that PAE4 to PAE6 are configured so that they respond to events differently from the other PAEs. For example, in Fig. 13g*), H is activated instead of A in response to event I2 (see Fig. 13g). The same holds true for 13h*). Instead of loading G in response to event I3 in Fig.
25   13i*), configuration F remains preserved and A is activated. In Fig. 13j*), it is shown for PAE7 that event I3 will again cause G to be loaded. In PAE4, event I4 causes F to be activated instead of configuration G (see Fig. 13j).

30   In Fig. 13, a wave of reconfigurations moves in response to events through a number of PAEs, which may have a two- or multidimensional design.

       It is not absolutely necessary that a reconfiguration having
35   taken place once take place throughout the entire flow. For example, reconfiguration with activation of A in response to event (I2) could take place only locally in PAEs 1 to 3 and PAE7, while configuration H continues to remain activated in

all the other PAEs.

In other words:

5   a)   It is possible that an event only occurs locally and
           therefore has only local reactivation as a result;
   b)   a global event may not have any effect on some PAEs,
           depending on the algorithm being executed.

10   In PAEs which continue to keep H activated even after (12),
the receipt of event (13) may, of course, have a completely
different effect, (I) such as activation of C instead of
loading of G; (ii) also, (13) might not have any effect at all
on these PAEs.

15

The Processor Model

The graphs shown in the following figures always have **[one]**
module as a graph node, it being assumed that a plurality of
20   modules can be mapped onto one target module. This means that,
although all modules are time independent of one another,
reconfiguration is performed and/or a data storage device is
inserted only in those modules which are marked with a
vertical line and Δt. This point is referred to as
25   reconfiguration time.

The reconfiguration time depends on certain data or the states
resulting from the processing of certain data.

30   In summary, this means:

1. Large modules can be partitioned at suitable points and
broken down into small modules which are time independent of
one another, and fit into the PAE array in an optimum manner.
35   2. In the case of small modules, which can be mapped together
onto a target module, time independence is not needed. This
saves configuration steps and speeds up data processing.
3. The reconfiguration times are positioned according to the

resources of the target modules. This makes it possible to
scale the graph length in any desired manner.

4. Modules can be configured with superimposition.

5. The reconfiguration of modules is controlled through the

5   data itself or through the result of data processing.

6. The data generated by the modules is stored and the
chronologically subsequent modules read the data from this
memory and in turn store the results in a memory or output the
end result to the peripheral devices.

10

Status Information of the Processor Model

In order to determine the states within a graph, the status

15   registers of the individual cells (PAEs) are made available to
all the other arithmetic units via a freely routable and
segmentable status bus system (0802) which exists in addition
to the data bus (0801) (Fig. 8b). This means that a cell (PAE
X) can evaluate the status information of another cell (PAE Y)

20   and process the data accordingly. In order to show the
difference with respect to existing parallel computing
systems, Fig. 8a shows the related art in the form of a
multiprocessor system whose processors are connected to one
another via a common data bus (0803). No explicit bus system

25   exists for synchronized exchange of data and status.

In other words, the network of the status signals (0802)
represents a freely and specifically distributed status
register of a single conventional processor (or of multiple

30   processors of an SMP computer). The status of each individual
ALU (i.e., each individual processor) and, in particular, each
individual piece of status information is available to the ALU
or ALUs (processors) that need the information. There is no
additional program runtime or communication runtime (except

35   for the signal runtimes) for exchange of information between
the ALUs (processors).

In conclusion, it should be noted that, depending on the task,

both the data flow chart and the ·control flow chart can be
treated according to the above-described method.

Virtual Machine Model

5

According to the previous sections, the principles of data
processing using VPU modules are mainly data flow oriented.
However, in order to execute sequential programs with a
reasonable performance, a sequential data processing model
10   must be available for which the sequencers in the individual
PAEs are often insufficient.

However, the architecture of VPUs basically allows sequencers
of any desired complexity to be formed from individual PAEs.
15   This means:

1.   Complex sequencers which exactly correspond to the
        requirements of the algorithm can be configured;
2.   Through appropriate configuration, the data flow may
20   exactly represent the computing steps of the algorithm.

Thus a virtual machine corresponding in particular to the
sequential requirements of an algorithm can be implemented on
VPUs.

25

The main advantage of the VPU architecture is that an
algorithm can be broken down by a compiler so that the data
flow portions are extracted [and] represented by an "optimum"
data flow, in that an adjusted data flow is configured AND the
30   sequential portions of the algorithm are represented by an
"optimum" sequencer, by configuring an adjusted sequencer. A
plurality of sequencers and data flows can be accommodated on
one VPU at the same time, depending exclusively on the
available resources.

35

As a result of the large number of PAEs, there are a large
number of local states within a VPU during operation. When
changing tasks or calling a subprogram (interrupts), these

states must be saved (see PUSH/POP for standard processors). This, however, cannot be done in practice due to the large number of states.

5   In order to reduce the states to a manageable number, a distinction must be made between two types of state:

1.   Status information of the machine model (MACHINE-STATE). This status information is only valid within the

10   execution of a specific module and is also only used locally in the sequencers and data flow units of this specific module. This means that these MACHINE STATEs represent the states occurring in the background within the hardware in processors according to the related art,

15   are implicit in the commands and processing steps, and have no further information for subsequent commands after the execution of a command. Such states need not be saved. The condition for this is that interrupts should only be executed after the complete execution of all the

20   currently active modules. If interrupts for execution arise, no new modules are loaded, but only those still active are executed; moreover, if allowed by the algorithm, no new operands are sent to the active modules. Thus a module becomes an indivisible,

25   uninterruptible unit, comparable to an instruction of a processor according to the related art.

2.   States of data processing (DATA-STATE). The data-related states must be saved and written into the memory when an

30   interrupt occurs according to the related art processor models. These are specific required registers and flags or, according to the terminology of VPU technology, triggers.

35   In the case of DATA-STATEs, handling can be further simplified depending on the algorithm. Two basic strategies are explained in detail below:

1. Concomitant run of the status information

All the relevant status information that is needed at a later time is transferred from one module to the next as normally implemented in pipelines. The status information is then

5   implicitly stored, together with the data, in a memory, so that the states are also available when the data is called. Therefore, no explicit handling of the status information takes place, in particular using PUSH and POP, which considerably speeds up processing depending on the algorithm,

10   as well as results in simplified programming. The status information can be either stored with the respective data packet or, only in the event of an interrupt, saved and specifically marked.

15   2. Saving the reentry address

When large amounts of data stored in a memory are processed, it may be advantageous to pass the address of at least one of the operands of the data packet just processed together with the data packet through the PAEs. In this case the address is

20   not modified, but is available when the data packet is written into a RAM as a pointer to the operand processed last.

This pointer can be either stored with the respective data packet or, only in the event of an interrupt, can be saved and

25   specifically marked. In particular, if all pointers to the operands are computed using one address (or a group of addresses), it may be advantageous to save only one address (or a group of addressees).

30   **"ULIW"-"UCISC" Model**

In order to understand this model, which is very similar to a processor according to the related art, the concept of VPU architecture must be extended. The virtual machine model is

35   used as a basis. The array of PAEs (PA) is considered as an arithmetic unit with a configurable architecture. The CT(s) represent(s) a load unit (LOAD-UNIT) for opcodes. The IOAG(s) take over the bus interface and/or the register set.

This arrangement allows two basic modes of operation which can
be used mixed during operation:

1. A group of PAEs (which can also be one PAE) is configured
to execute a complex command or command sequence and then the
data associated with this command (which can be a single data
word) is processed. Then this group is reconfigured to process
the next command. The size and arrangement of the group may
change. According to partitioning technologies mentioned
previously, it is the compiler's responsibility to create
optimum groups to the greatest possible extent. Groups are
"loaded" as commands onto the module by the CT; therefore, the
method is comparable to the known VLIW, except that
considerably more arithmetic units are managed AND the
interconnection structure between the arithmetic units can
also be covered by the instruction word (Ultra Large
Instruction Word = "ULIW"). This allows a very high
Instruction Level Parallelism (ILP) to be achieved. (See also
Fig. 27.) One instruction word corresponds here to one module.
A plurality of modules can be processed simultaneously, as
long as the dependence of the data allows this and sufficient
resources are available on the module. As in the case of VLIW
commands, usually the next instruction word is immediately
loaded after the instruction word has been executed. In order
to optimize the procedure in terms of time, the next
instruction word can be pre-loaded even during execution (see
Fig. 10). In the event of a plurality of possible next
instruction words, more than one can be pre-loaded and the
correct instruction word is selected prior to execution, e.g.,
by a trigger signal. (See Fig. 4a B1/B2, Fig. 15 ID C/ID K,
Fig. 36 A/B/C.)

2. A group of PAEs (which can also be one PAE) is configured
to execute a frequently used command sequence. The data, which
can also in this case be a single data word, is sent to the
group as needed and received by the group. This group remains
without being reconfigured for a plurality of cycles. This
arrangement is comparable with a special arithmetic unit in a

14   **LITERAL TRANSLATION**

processor according to the related art (e.g., MMX), which is
provided for special tasks and is only used as needed. With
this method, special commands can be generated according to
the CISC principle with the advantage that these commands can
5   be configured to be application-specific (Ultra-CISC = UCISC).

Extension of the RDY/ACK Protocol (see PACT02)

PACT02 describes a RDY/ACK standard protocol, which describes
10   the essential requirements according to the synchronization
procedures of German Patent 44 16 881 with respect to a
typical data flow application. The disadvantage of the
protocol is that only data can be transmitted and receipt
acknowledged. Although the reverse case, with data being
15   requested and transmission acknowledged (hereinafter referred
to as REQ/ACK), can be implemented electrically with the same
two-wire protocol, it is not detected semantically. This is
particularly true when REQ/ACK and RDY/ACK are used in mixed
operation.
20
Therefore, a clear distinction is made between the protocols:
RDY: data is available at the transmitter for the receiver;
REQ: data is requested by the receiver from the transmitter;
ACK: general acknowledgment for receipt or transmission
25   completed

(In principle, a distinction could also be made between ACK
for a RDY and an ACK for a REQ, but the semantics of the ACK
is usually implicit in the protocols.)
30
Memory Model

Memories (one or more) can be integrated in VPUs and addressed
as in the case of a PAE. In the following, a memory model
35   shall be described which represents at the same time an
interface to external peripherals and/or external memories:

A memory within a VPU with PAE-like bus functions may

represent various memory modes:

1.   Standard memory (random access)

2.   Cache (as an extension of the standard memory)

5   3.   Lookup table

4.   FIFO

5.   LIFO (stack).

A controllable interface, which writes into or reads from

10   memory areas either one word or one block at a time, is

associated with the memory.

The following usage options result:

15   1.   Isolation of data streams (FIFO)

2.   Faster access to selected memory areas of an external
     memory, which represents a cache-like function (standard
     memory, lookup table)

3.   Variable-depth stack (LIFO).

20

The interface can be used, but it is not absolutely necessary

if, for example, the data is used only locally in the VPU and

the free memory in an internal memory is sufficient.

25   **Stack Model**

A simple stack processor can be designed by using the REQ/ACK

protocol and the internal memory in the LIFO mode. In this

mode, temporary data is written by the PAEs to the stack and

30   loaded from the stack as needed. The necessary compiler

technologies are sufficiently known. The stack can be as large

as needed due to the variable stack depth, which is achieved

through a data exchange of the internal memory with an

external memory.

35

Accumulator Model

Each PAE can represent an arithmetic unit according to the

accumulator principle. As is known from PACT02, the output register can be looped back to the input of the PAE. This yields a related art accumulator. Simple accumulator processors can be designed in connection with the sequencer

5   according to Fig. 11.

Register Model

A simple register processor can be designed by using the

10   REQ/ACK protocol and the internal memory in the standard memory mode. The register addresses are generated by one group of PAEs, while another group of PAEs is responsible for processing the data.

15   **Memory Architecture**

The memory has two interfaces: a first interface which connects the memory to the array, and a second one which connects the memory with an IO unit. In order to improve the

20   access time, the memory should be designed as a dual-ported RAM, which allows read and write accesses to take place independently of one another.

The first interface is a conventional PAE interface (PAEI),

25   which guarantees access to the bus system of the array and ensures synchronization and trigger processing. Triggers can be used **[in order to]** display different states of the memory or to force actions in the memory, for example,

30   1.   Empty/full: when used as a FIFO, the FIFO status "full," "almost full," "empty," or "almost empty" is displayed;
     2.   Stack overrun/underrun: when used as a stack, stack overrun and underrun are signaled;
     3.   Cache hit/miss: in the cache mode, it is displayed whether

35   an address has been found in the cache;
     4.   Cache flush: writing the cache into the external RAM is forced by a trigger.

A configurable state machine, which controls the different operating modes, is associated with the PAE interface. A counter is associated with the state machine in order to generate the addresses in FIFO and LIFO modes. The addresses

5   are supplied to the memory via a multiplexer, so that additional addresses generated in the array can be supplied to the memory.

The second interface is used to connect an IO unit (IOI). The

10   IO unit is designed as a configurable controller having an external interface. The controller reads or writes data one word or one block at a time from and into the memory. The data is exchanged with the IO unit. The controller also supports different cache functions using an additional TAG memory.

15

IOI and PAEI are synchronized with one another, so that no collision of the two interfaces can occur. Synchronization is different depending on the mode of operation; for example, while in standard memory or stack mode operation either the

20   IOI or the PAEI may access the entire memory at any time, synchronization is row by row in the FIFO mode, i.e., while IOI accesses a row x, the PAEI can access any other row other than x at the same time.

25   The IO unit is configured according to the peripheral requirements, for example:
1.   SDRAM controller
2.   RDRAM controller
3.   DSP bus controller
30   4.   PCI controller
5.   serial controller (e.g., NGIO)
6.   special purpose controller (SCSI, Ethernet, USB, etc.).

A VPU may have any desired memory elements having any desired

35   IO units. Different IO units can be implemented in a single VPU.

Mode of Operation:

1. Standard memory

1.1 internal/local

Data and addresses are exchanged with the memory via the PAEI.
The addressable memory size is limited by the size of the

5   memory.

1.2 external/memory mapped window

Data and addresses are exchanged with the memory via the PAEI.
A base address in the external memory is specified in the IOI

10   controller. The controller reads data from the external memory
address one block at a time and writes it into the memory, the
internal and external addresses being incremented (or
decremented) with each read or write operation, until the
entire internal memory has been transmitted or a predefined

15   limit has been reached. The array works with the local data
until the data is written again into the external memory by
the controller. The write operation takes place similarly to
the read operation described previously.

20   Read and write by the controller can be initiated
a) by a trigger or
b) by access of the array to an address that is not locally
stored. If the array accesses such an address, initially the
internal memory is written to the external one and then the

25   memory block is reloaded with the desired address.

This mode of operation is particularly relevant for the
implementation of a register set for a register processor. In
this case, the push/pop of the register set with the external

30   memory can be implemented using a trigger for a change in task
or a context switchover.

1.3 external/lookup table

The lookup table function is a simplification of 1.2. In this

35   case, the data is read once or a number of times via a CT call
or a trigger from the external RAM into the internal RAM. The
array reads data from the internal memory, but writes no data
into the internal memory. The base address in the external

memory is stored in the controller either by the CT or by the
array and can be modified at runtime. Loading from the
external memory is initiated either by the CT or by a trigger
from the array and can also be done at runtime.

5

1.4 external/cached

In this mode, the array optionally accesses the memory. The
memory operates as a cache memory for the external memory
according to the related art. The cache can be emptied (i.e.,
10   the cache can be fully written into the external memory)
through a trigger from the array or through the CT.

2. FIFO

The FIFO mode is normally used when data streams are sent from
15   the outside to the VPU. Then the FIFO is used to isolate the
external data processing from the data processing within the
VPU so that either the write operation to the FIFO takes place
from the outside and the read operation is performed by the
VPU or vice versa. The states of the FIFO are signaled by
20   triggers to the array or, if needed, also to the outside. The
FIFO itself is implemented according to the related art with
different read and write pointers.

3. Stack/internal

25   An internal stack is formed by an address register. The
register is (a) incremented or (b) decremented depending on
the mode with each write access to the memory by the array. In
contrast, in the case of read accesses from the array, the
register is (a) decremented and (b) incremented. The register
30   makes the required address available for each access. The
stack is limited by the size of the memory. Errors
(overrun/underrun) are indicated by triggers.

4. Stack/external

35   If the internal memory is too small for forming a stack, it
can be transferred into the external memory. For this purpose,
an address counter for the external stack address is available
in the controller. If a certain amount of records is exceeded

in the internal stack, a number of records is written onto the external stack one block at a time. The stack is written outward from the end, i.e., from the oldest record, a number of the newest records not being written to the external

5   memory, but remaining internal. The external address counter (ERC) is modified one row at a time.

After space has been created in the internal stack, the remaining content of the stack must be moved to the beginning

10   of the stack; the internal stack address is adjusted accordingly.

A more efficient version is configuring the stack as a ring memory (see PACT04). An internal address counter is modified

15   by adding or removing stack entries. As soon as the internal address counter (IAC) exceeds the top end of the memory, it points to the lowermost address. If the IAC is less than the lowermost address, it points to the uppermost one. An additional counter (FC) indicates the full status of the

20   memory, i.e., the counter is incremented with each word written, and decremented with each word read. Using the FC, it can be ascertained when the memory is full or empty. This technology is known from FIFOs. Thus, if a block is written into the external memory, the adjustment of the FC is

25   sufficient for updating the stack. An external address counter (EAC) always points to the oldest record in the internal memory and is therefore at the end of the stack opposite the IAC. The EAC is modified if

(a)   data is written to the external stack; then it runs

30   toward the IAC;

(b)   data is read from the external stack; then it moves away from the IAC.

It is ensured by monitoring the FC that IAC and EAC do not

35   collide.

The ERC is modified according to the external stack operation (buildup or reduction).

MMU

A MMU can be associated with the external memory interface.
The MMU performs two functions:

5

1. Recomputes the internal addresses to external addresses in
order to support modern operating systems;
2. Monitors accesses to the external addresses, e.g.,
generates an error signal as a trigger if the external stack

10 overruns or underruns.

Compiler

The VPU technology programming principle according to the

15 present invention includes separating sequential codes and
breaking them down into the largest possible number of small
and independent subalgorithms, while the subalgorithms of the
data flow code **[are]** mapped directly onto the VPU.

20 **Separation between VPU code and standard code**
C++ will be used in the following to represent all possible
compilers (Pascal, Java, Fortran, etc.) within a related art
language; a special extension (VC = VPU C), which contains the
language constructs and types which can be mapped onto VPU

25 technology particularly well, can be defined. VCs may be used
by programmers only within methods or functions that use no
other constructs or types. These methods and functions can be
mapped directly onto the VPU and run particularly efficiently.
The compiler extracts the VC in the pre-processor and forwards

30 it directly to the VC back-end processing (VCBP).

Extraction of the parallelizable compiler code
In the following step, the compiler analyzes the remaining C++
codes and extracts the portions (MC = mappable C) which can be

35 readily parallelized and mapped onto the VPU technology
without the use of sequencers. Each individual MC is placed
into a virtual array and routed. Then the space requirement

and the expected performance are analyzed. For this purpose, the VCBP is called and the individual MCs are partitioned together with the VCs, which are mapped in each case.

5   The MCs whose VPU implementations achieve the highest increase in performance are accepted and the others are forwarded to the next compiler stage as C++.

Optimizing Sequencer Generator

10

This compiler stage can be implemented in different ways depending on the architecture of the VPU system:

1.   VPU without a sequencer or external processor
15   All remaining C++ codes are compiled for the external processor.

2.   VPU only with sequencer
2.1.   Sequencer in the PAEs
20   All remaining C++ codes are compiled for the sequencer of the PAEs.
2.2   Configurable sequencer in the array
The remaining C++ code is analyzed for each independent module. The best-suited sequencer version is selected from a
25   database and stored as VC code (SVC). This step is mostly iterative, i.e., a sequencer version is selected, the code is compiled, analyzed, and compared to the compiled code of other sequencer versions. Finally, the object code (SVCO) of the C++ code is generated for the selected SVC.
30   2.3   Both 2.1 and 2.2 are used
The mode of operation corresponds to that of 2.2. Special static sequencer models are available in the database for the sequencers in the PAEs.

35   3.   VPU with sequencer and external processor
This mode of operation also corresponds to 2.2. Special static sequencer models are available in the database for the external processor.

Linker

The linker connects the individual modules (VC, MC, SVC, and SVCO) to form an executable program. For this purpose, it uses the VCBP in order to place and route the individual modules

5   and to determine the time partitioning. The linker also adds the communication structures between the individual modules and, if needed, adds registers and memories. Structures for storing the internal states of the array and sequencers for the case of a reconfiguration are added on the basis of an

10   analysis of the control structures and dependencies of the individual modules.

Notes on the processor models

The machine models used can be combined within a VPU in any

15   desired manner. It is also possible to switch from one model to another within an algorithm depending on which model is best.

If an additional memory is added to a register processor from

20   which the operands are read and into which the results are written, a load/store processor can be created. A plurality of different memories can be assigned by treating the individual operands and the result separately.

25   These memories then operate more or less as load/store units and represent a type of cache for the external memory. The addresses are computed by the PAEs which are separate from the data processing.

30   **Pointer reordering**

High-level languages such as C/C++ often use pointers, which are poorly handled by pipelines. If a pointer is not computed until immediately before a data structure at which it points is used, the pipeline often cannot be filled rapidly enough

35   and the processing is inefficient, especially in the VPUs.

It is certainly useful not to use any pointers if possible in programming VPUs; however, this is often impossible.

The problem is solved by having the pointer structures re-sorted by the compiler so that the pointer addresses are computed as early as possible before they are used. At the same time, there should be as little direct dependence as

5   possible between a pointer and the data at which it points.

Extensions of the PAEs (compared to Patents 196 51 075.9 and 196 54 846.2

10   Patents 196 51 075.9 and 196 54 846.2 define the related art with respect to the configuration characteristics of cells (PAEs).

We shall discuss two characteristics in detail:

15

1. According to Patent 196 51 075.9, a set of configuration registers (0904) containing a configuration is associated with a PAE (0903) (Fig. 9a);

2. According to Patent 196 54 846.2, a group of PAEs (0902)

20   can access a memory to store or read data (Fig. 9b).

The object is
a) to provide a method to speed up the reconfiguration of PAEs and isolate it in time from the higher-level load unit, and

25   b) to design the method so that the possibility of simultaneously sequencing over more than one configuration is provided, and
c) to simultaneously hold in one PAE a plurality of configurations, one of which is always activated, with rapid

30   switching between different configurations.

Isolation of the configuration register
The configuration register is isolated from the higher-level load unit (CT) (Fig. 10) by the use of a set of a plurality of

35   configuration registers (1001). Precisely one of the configuration registers always selectively determines the function of the PAE. The active register is selected via a multiplexer (1002). The CT can freely write into each of the

configuration registers as long as the configuration register does not determine the current configuration of the PAE, i.e., is not active. Writing onto the active register is possible in principle, using the method described in PACT10.

5

The configuration register to be selected by 1002 can be determined by different sources:

1. Any status signal or a group of any status signals

10 supplied via a bus system (0802) to 1002 (Fig. 10a). The status signals are generated by any of the PAEs or made available through external links of the module (see Fig. 8).
2. The status signal of the PAE which is configured by 1001/1002 is used for the selection (Fig. 10b).

15 3. A signal generated by the higher-level CT is used for the selection (Fig. 10c).

Optionally, the incoming signals (1003) can be stored for a certain period of time using a register and can be optionally

20 called as needed.

By using a plurality of registers, the CT is isolated in time. This means that the CT can "pre-load" a plurality of configurations without a direct time-dependency existing.

25

The configuration of the PAE is only delayed until the CT has loaded the register if the selected/activated register in 1001 has not yet been loaded. In order to determine whether a register has valid information, a "valid bit" (1004) which is

30 set by the CT can be inserted in each register, for example. If 0906 is not set in a selected register, CT is requested, via a signal, to configure the register as rapidly as possible.

35 The method described in Fig. 10 can be easily extended to a sequencer (Fig. 11). For this purpose, a sequencer having an instruction decoder (1101) is used for triggering the selection signals of the multiplexer (1002). The sequencer

determines, as a function of the currently selected configuration (1102) and an additional piece of status information (1103/1104) the configuration to be selected next. The status information can be

5   (a) the status of the status signal of the PAE which is configured by 1001/1002 (Fig. 11a);
(b) any desired status signal supplied via 0802 (Fig. 11b);
(c) a combination of (a) and (b).

10   1001 can also be designed as a memory, with a command being addressed by 1101 instead of 1002. Addressing here depends on the command itself and on a status register. In this respect, the structure corresponds to that of a "von Neumann" machine with the difference

15   (a) of universal applicability, i.e., non-use of the sequencer (see Fig. 10);
(b) that the status signal does not need to be generated by the arithmetic unit (PAE) associated with the sequencer, but may come from any other arithmetic unit (see Fig. 11b).

20

It is important that the sequencer can execute jumps, in particular also conditional jumps within 1001.

Another additional or alternative method (Fig. 12) for
25   creating sequencers within VPUs is the use of the internal data storage device (1201, 0901) for storing the configuration information for a PAE or a group of PAEs. In this case, the data output of a memory is connected to a configuration input or data input of a PAE or a plurality of PAEs (1202). The
30   address (1203) for 1201 can be generated by the same PAE/PAEs or any one or more other PAE(s).

In this method, the sequencer is not fixedly implemented, but is emulated by a PAE or a group of PAEs. The internal memories
35   can reload programs from the external memories (see memory system according to the present invention).

In order to store local data (e.g., for iterative computations

          LITERAL TRANSLATION

and as a register for a sequencer), the PAE is provided with
an additional register set, whose individual registers are
either determined by the configuration, connected to the ALU
or written into by the ALU; or they be freely used by the
5   command set of an implemented sequencer (register mode). One
of the registers can also be used as an accumulator
(accumulator mode). If the PAE is used as a full-featured
machine, it is advantageous to use one of the registers as an
address counter for external data addresses.

10

In order to manage stacks and accumulators outside the PAE
(e.g., in the memories according to the present invention),
the previously described RDY/ACK REQ/ACK synchronization model
is used.

15

Related art PAEs (see PACT02) are ill-suited for processing
bit-wise operations, since the integrated ALU does not
particularly support bit operations, i.e., it has a narrow
design (1, 2, 4 bits wide). Efficient processing of individual
20   bits or signals can be guaranteed by replacing the ALU core
with an FPGA core (LC), which executes logical operations
according to its configuration. The LC can be freely
configured in its function and internal interconnections.
Related art LCs can be used. For certain operations it is
25   advantageous to assign a memory to the LC internally. The
interface modules between FC and the bus system of the array
are only adjusted slightly to the FC, but are basically
preserved. However, in order to configure the time response of
the FC in a more flexible manner, it is useful if the
30   registers in the interface modules are configured so that they
can be turned off.

Figures

Fig. 4a shows some basic characteristics of the method according to the present invention:

The Type A modules are combined into a group and, at the end,

5 have a conditional jump either to B1 or to B2. At this position (0401), a reconfiguration point is inserted, since it is useful to treat each branch of the conditional jump as a separate group (case 1). However, if both B branches (B1 and B2), together with A as well, suit the target module (case 2),

10 it is more convenient to insert only one reconfiguration point at 0402, since this reduces the number of configurations and increases the processing speed. Both branches (B1 and B2) jump to C at 0402.

15 The configuration of cells on the target module is schematically shown in Fig. 4b. The functions of the individual graph nodes are mapped onto the cells of the target module. Each line represents one configuration. The broken-line arrows at a new line indicate a reconfiguration. S_n is a

20 data storage cell of any desired design (register, memory, etc.). $S_n I$ is a memory which accepts data and $S_n O$ is a memory which outputs data. Memory S_n is always the same for the same n; I ad O identify the direction of data transfer.

25 Both cases of conditional jump (case 1, case 2) are shown.

The model of Fig. 4 corresponds to a data flow model, however, with the important extension of the reconfiguration point and the graph partitioning that is achieved thereby, the data

30 transmitted between the partitions being buffered.

In the model of Fig. 5a, a graph B_n is called from a number of graphs B of any number and combination of graphs (0501). After execution of B, the data is returned to 0501.

35

If a sufficiently large sequencer (A) is implemented in 0501, a principle which is very similar to typical processors can be implemented with this model. In this case, the data goes to

1.   sequencer A, which decodes it as commands and responds to
it according to the "von Neumann" principle;
2.   sequencer A, where it is treated as data and forwarded to
a fixedly configured arithmetic unit C for computation.

5

Graph B selectively makes available a special arithmetic unit
and/or special opcodes for certain functions and is
alternatively used to speed up C. For example, B1 can be an
optimized algorithm for performing matrix multiplications,
10   while B2 represents a FIR filter, and B3 a pattern
recognition. The appropriate, i.e., corresponding graph B is
called according to an opcode which is decoded by 0501.

Fig. 5b schematically shows the mapping onto the individual
15   cells, the pipeline-type arithmetic unit character being
symbolized in 0502.

While preferably larger memories are introduced at the
reconfiguration points of Fig. 4 for temporary storage of
20   data, simple synchronization of data is sufficient at the
reconfiguration points of Fig. 5, since the data stream
preferably runs as a whole through graph B, and graph B is not
partitioned further; therefore, temporary storage of data is
superfluous.

25

Fig. 6a shows different loops. Loops can be basically handled
in three different ways:

1.   Hardware approach: Loops are mapped onto the target
30   hardware completely rolled out (0601a/b). As explained
previously, this is only possible for a few types of loops;
2.   Data flow approach: Loops are formed over a plurality of
cells within the data flow (0602a/b). The end of the loop is
looped back to the beginning of the loop.
35   3.   Sequencer approach: A sequencer having a minimum command
set executes the loop (0603a/b). The cells of the target
modules are configured so that they contain the corresponding
sequencers (see Fig. 11a/b).

The execution of the loops can sometimes be optimized by
breaking them down in a suitable manner:

1. Using related art optimizing methods, often the body of
the loop, i.e., the part to be executed repeatedly, can be
optimized by removing certain operations from the loop and
placing them before or after the loop (0604a/b). Thus the
number of commands to be sequenced is substantially reduced.
The removed operations are only executed once before or after
the execution of the loop.
2. Another optimization option is dividing the loops into a
plurality or smaller or shorter loops. This division is
performed so that a plurality of parallel or sequential
(0605a/b) loops are obtained.

Fig. 7 illustrates the implementation of a recursion. The same
resources (0701) are used in the form of cells for each
recursion level (1 - 3). The results of each recursion level
(1 - 3) are written into a stack-type memory (0702) as it is
being built up (0711:). The stack is torn down simultaneously
with the tear-down (0712:) of the levels.

Fig. 14 shows the virtual machine model. Data (1401) and
states (1402) associated with the data are read into a VPU
(1403) from an external memory. 1401/1402 are selected via an
address 1404 generated by the VPU. PAEs are combined to form
different groups within the VPU (1405, 1406, 1407). Each group
has a data processing part (1408), which has local implicit
states (1409), which have no effect on the surrounding groups.
Therefore the states of the data processing part are not
forwarded outside the group. However, it may depend on the
external states. Another part (1410) generates states which
have an effect on the surrounding groups.

The data and states of the results are stored in another
memory (1411, 1412). At the same time, the address of operands
(14004) can be stored as a pointer (1413). 1404 can pass
through registers (1414) for time synchronization.

NY01 419288   31   **LITERAL TRANSLATION**

Fig. 14 shows a simple model for the sake of clarity. The interconnection and grouping may be considerably more complex than they are in this model. States and data can also be transmitted to modules other than those mentioned below. Data is transmitted to different modules than the states. Both data and states of a certain module may be received by a plurality of different modules. 1408, 1409, and 1410 may be present within a group. Depending on the algorithm, individual parts may also not be present (e.g., 1410 and 1409 present, but not 1410).

Figure 15 shows how subapplications are extracted from a processing graph. The graph is broken down so that long graphs are subdivided into smaller parts as appropriate and mapped in subapplications (H, A, C, K). After jumps, new subgraph s are formed (C, K), with a separate subgraph being formed for each jump.

In the ULIW model, each subgraphgraph can be loaded separately by the CT (see PACT10). It is essential that subgraphs can be managed by the mechanisms of PACT10. These include in particular intelligent configuring, execute/start, and deletion of subapplications.

1503 causes subapplication A to be loaded or configured, while subapplication K is being executed. Thus,
a) subapplication A is already configured in the PAEs at the time subapplication K is completely executed if the PAEs have more than one configuration register;
b) subapplication A is already loaded into the CT at the time subapplication K is completely executed if the PAEs only have one configuration register.
1504 starts the execution of subapplication K.
This means that, at runtime the next required program parts are loaded independently while the current program parts are running. This yields a much more efficient handling of the program codes than the usual cache mechanisms.

32   **LITERAL TRANSLATION**

Another particular feature of subapplications A is shown. In principle, both possible branches (C, K) of the comparison could conceivably be preconfigured. Assuming that the number of free configuration registers available is insufficient for

5 this, the more probable of the two branches is configured (1506). This also saves configuration time. When the non-configured branch is executed, the program execution is interrupted (since the configuration is not yet loaded into the configuration registers) until the branch is configured.

10

In principle, unconfigured subapplications can also be executed (1505); in this case they must be loaded prior to execution as described previously.

15 A FETCH command can be initiated by a trigger via its own ID. This allows subapplications to be pre-loaded depending on the status of the array.

The ULIW model essentially differs from the VLIW model in that

20 it
1. also includes data routing;
2. forms larger instruction words.

The above-described partitioning method can also be used by

25 compilers for existing standard processors according to the RISC/CISC principle. In that case, if a unit (CT) according to PACT10 is used for controlling the command cache, it can be substantially optimized and sped up.

30 For this purpose, "normal" programs are partitioned into subapplications in an appropriate manner. According to PACT10, references to possible subsequent subapplications are inserted (1501, 1502). Thus a CT can pre-load the subapplications into the cache before they are needed. In the case of a jump, only

35 the subapplication to which the jump was made is executed; the other(s) are overwritten later by new subapplications. In addition to intelligent pre-loading, the method has the additional advantage that the size of the subapplications is

already known at the time of loading. Thus optimum bursts can be executed by the CT when accessing the memories, which in turn considerably speeds up memory access.

Figure 16 shows the structure of a stack processor. Protocols are generated by the PAE array (1601) in order to write into or read from a memory (1602) configured as LIFO. A RDY/ACK protocol is used for writing and a REQ/ACK protocol is used for reading. The interconnection and operating modes are configured by the CT (1603). 1602 can transfer its content to the external memory (1604).

An array of PAEs operates as a register processor in this embodiment (Figure 17). Each PAE is composed of an arithmetic unit (1701) and an accumulator (1702) to which the result of 1701 is looped back (1703). Thus, in this embodiment, each PAE represents an accumulator processor. A PAE (1705) reads and writes the data into the RAM (1704) configured as a standard memory. An additional PAE (1706) generates the register addresses.

Often it is advisable to use a separate PAE for reading the data. Then 1705 would only write and PAE 1707 would only read. An additional PAE (1708, shown in broken lines underneath) is to be added for generating the read addresses.

It is not absolutely necessary to use separate PAEs for generating addresses. Often the registers are implicit and, configured as constants, can then be transmitted by the data processing PAEs.

The use of accumulator processors for a register processor is shown as an example. PAEs without accumulators can also be used for creating register processors. The architecture shown in Figure 17 can be used for activating registers as well as for activating a load/store unit.

When used as a load/store unit, it is almost obligatory to

LITERAL TRANSLATION

connect an external RAM (1709) downstream, so that 1704 only represents a temporary section of 1709, similar to a cache.

Also when 1704 is used as a register bank, it is to some
5   extent recommended that an external memory be connected downstream. Thus PUSH/POP operations according to the related art, which write the content of the register into a memory or read it from there, can be performed.

10 Figure 18 shows a complex machine as an example, in which the PAE array (1801) controls a load/store unit (1802) with a downstream RAM (1803), and also has a register bank (1804) with a downstream RAM (1805). 1802 and 1804 can be activated by one PAE each or any group of PAEs. The unit is controlled
15 by a CT (1806) according to the VPU principle.

It is important that there is no basic difference between the load/store unit (1802) and the register bank (1804) and their activation.
20

Figures 19, 20, 21 show an internal memory according to the present invention, which at the same time represents a communication unit having external memories and/or peripheral devices. The individual figures show different modes of
25 operation of the same memory. The modes of operation and the individual detail settings are configured.

Figure 19a shows a memory according to the present invention in the "register/cache" mode. In the memory (1901) according
30 to the present invention data words of a usually larger and slower external memory (1902) are stored.

The data exchange between 1901, 1902, and the PAEs (not shown) connected via a bus (1903) takes place as follows, distinction
35 being made between two modes of operation:

A) The data read or transmitted by the PAEs from main memory 1902 is buffered in 1901 using a cache technique. Any known

cache technique can be used.

B) The data of certain addresses is transmitted between 1902 and 1901 via a load/store unit. Certain addresses are

5   predefined both in 1902 and in 1901, different addresses being normally used for 1902 and 1901. The individual addresses can be generated by a constant or by computations in PAEs. In this operating mode memory 1901 operates as a register bank.

10   The addresses between 1901 and 1902 can be assigned in any desired manner, which only depends on the respective algorithms of the two operating modes.

     The corresponding machine is shown in Fig. 19b as a block

15   diagram. A control unit (1904) operating as a load/store unit (1904) (according to the related art) or as a cache controller (according to the related art) is associated with the bus between 1901 and 1902. If necessary, a memory management unit (MMU) (1905) with address translation and address checking can

20   be associated with this unit. Both 1904 and 1905 can be activated by the PAEs. Thus, for example, the MMU is programmed, the load/store addresses are set, or a cache flush is triggered.

25   Figure 20 shows the use of the memory (2001) in the FIFO mode, in which data streams are isolated according to the known FIFO principle. The typical application is in a write (2001a) or read (2001b) interface, in which case data is isolated in time between the PAEs connected to the internal bus system (2002)

30   and the peripheral bus (2003).

     A unit (2004) which controls the write and read pointers of the FIFO as a function of the bus operations of 2003 and 2002 is provided to control the FIFO.

35

     Figure 21 shows the operating principle of the memories according to the present invention in the stack mode. A stack (according to the related art) is a memory whose

uppermost/lowermost element is the one active at the time. Data is always appended at the top/bottom, and data is likewise removed from the top/bottom. This means that the data written last is also the data read first (last in first out).

5   It is irrelevant whether a stack grows upward or downward and it depends on the implementation. In the following embodiment, stacks growing upward will be discussed.

The current data is held in internal memory 2101; the most
10   recent record (2107) is located at the very top in 2101. Old records are transferred to external memory 2102. If the stack continues to grow, the space in internal memory 2101 is no longer sufficient. When a certain amount of data is reached, which may be represented by a (freely selectable) address in
15   2101 or a (freely selectable) value in a record counter, part of 2101 is written as a block to the more recent end (2103) of the stack in 2102. This part is the oldest and thus the least current data (2104). Subsequently, the remaining data in 2101 is shifted so that the data in 2101 copied to 2102 is
20   overwritten with the remaining data (2105) and thus sufficient free memory (2106) is created for new stack inputs.

If the stack decreases, starting at a certain (freely selectable) point, the data in 2101 is shifted so that free
25   memory is created after the oldest and least current data. A memory block is copied from 2102 into the freed memory, and is then deleted in 2102.

In other words, 2101 and 2102 represent a single stack, the
30   current records being located in 2101 and the older and less current records being transferred to 2102. The method represents a quasi-cache for stacks. The data blocks are preferably transmitted by block operations; therefore, the data transfer between 2101 and 2102 can be performed in the
35   rapid burst operating modes of modern memories (SDRAM, RAMBUS, etc.).

It should be mentioned again that in the embodiment of Figure

21 the stack grows upward. If the stack grows downward (a
frequently used method), the positions top/bottom and the
directions in which the data is moved within a memory are
exactly reversed.

5

Internal stack 2101 is preferably designed as a type of ring
memory. The data at one end of the ring is transmitted between
PAEs and 2101 and at the other end of the ring between 2101
and 2102. This has the advantage that data can be easily
10   shifted between 2101 and 2102 without having any effect on the
internal addresses in 2101. Only the position pointers of the
bottom and top data and the fill status counter have to be
adjusted. The data transfer between 2101 and 2102 can by
triggered by the known ring memory flags "almost full"/"full"
15   "almost empty"/"empty."

The required hardware is shown as a block diagram in Fig. 21b.
A unit (2110) for managing the pointers and the counter is
associated with internal stack 2101. A unit (2111) for
20   controlling the data transfers is looped into the bus (2114)
between 2101 and 2102. An MMU (2112) according to the related
art having the corresponding test systems and address
translations can be associated with this unit.

25   The connection between the PAEs and 2101 is implemented by bus
system 2113.

Figure 22 shows an example for the re-sorting of graphs. The
left-hand column (22..a) shows an unoptimized arrangement of
30   commands. Pointers A (2207a) and B (2211a) are loaded. One
cycle later in each case, the values of the pointers are
needed (2208a, 2212a). This dependence is too short to be
executed efficiently, since a certain time (2220a, 2221a) is
needed for loading from the memory. The time periods are
35   increased to a maximum (2220b, 2221b) by re-sorting the
commands (22..b). Although the value of the pointer of A is
needed in 2210 and 2208, 2208 is placed after 2210, since more
time is gained in this way for computing B. Computations that

are independent of pointers (2203, 2204, 2206) can be inserted between 2211 and 2212, for example, in order to gain more time for memory accesses. A compiler or assembler can perform the corresponding optimization using system parameters which

5   represent the access times.

Figure 23 shows the special case of Figures 4 - 7. An algorithm is often composed of data flow portions and sequential portions even within loops. Such structures can be

10   efficiently constructed according to the above-described method using the bus system described in PACT07. For this purpose, the RDY/ACK protocol of the bus system is initially extended by the REQ/ACK protocol according to the present invention. Thus register contents of individual PAEs can be

15   specifically queried by one or more other PAEs or by the CT. A loop (2305) is now broken down into at least two graphs: a first one (2301) which represents the data flow portion, and a second one (2302), which represents the sequential portion.

20   A conditional jump chooses one of the two graphs. The special characteristic is that now 2302 needs to know the internal status of 2301 for execution and vice versa, 2301 must know the status of 2302.

25   This is implemented by storing the status just once, namely in the registers of the PAEs of the higher-performance data flow graph (2301).

If a jump is performed in 2302, the sequencer reads, if

30   required, the states of the respective registers from (2303) using the bus system of PACT07. The sequencer performs its operations and writes all the modified states back (2304) into the registers (again via the bus system of PACT07). Finally, it should be mentioned that the above-mentioned graphs need

35   not necessarily be narrow loops (2305). The method is generally applicable to any subalgorithm which is executed multiple times within a program run (reentrant) and is run either sequentially or in parallel (data flow type); the

states must be transferred between the sequential and the parallel portions.

Wave reconfiguration offers considerable advantages regarding the speed of reconfiguration, in particular for simple sequential operations.

One basic characteristic of this processing method is that the sequencer can also be designed as an external microprocessor. This means that a processor is connected to the array via the data channels and exchanges local, temporary data with the array via bus systems. All sequential portions of an algorithm that cannot be mapped into the array of PAEs are run on the processor.

A distinction must be made between three bus systems:

1. Data bus which regulates the exchange of processed data between the VPU and the processor;
2. Register bus which enables access to the VPU registers and thus guarantees the data exchange (2302, 2304) between 2302 and 2301;
3. Configuration data bus, which configures the VPU array.

Figure 24 shows the effects over time.

Single-hatched areas represent data processing PAEs, 2401 showing PAEs after reconfiguration and 2403 showing PAEs before reconfiguration. Cross-hatched areas (2402) show PAEs which are being reconfigured or are waiting for reconfiguration.

Figure 24a shows the effect of wave reconfiguration on a simple sequential algorithm. Here it is possible exactly to reconfigure precisely those PAEs that have been assigned a new task. This can be performed efficiently, i.e., simultaneously, because a PAE receives a new task in each cycle.

A row of PAEs from the matrix of all PAEs of a VPU is shown as an example. The states in the cycles after cycle t are given with a one-cycle delay.

5   Figure 24b shows the effect over time of the reconfiguration of large portions. A number of PAEs of a VPU is shown as an example. The states in the cycles after cycle t are given with different delays of a plurality of cycles.

10   While initially only a small portion of the PAEs is being reconfigured or is waiting for reconfiguration, this area becomes larger over time until all PAEs are reconfigured. The enlarging of the area means that, due to the time delay of the reconfiguration, more and more PAEs will be waiting for
15   reconfiguration (2402), resulting in loss of computing capacity.

Therefore it is proposed that a wider bus system be used between the CT (in particular the memory of the CT) and the
20   PAEs, which provides sufficient lines for reconfiguring multiple PAEs at the same time within one cycle.

Figure 25 illustrates the scalability of the VPU technology. Scalability basically results from the rollout of a graph
25   without a time sequence separating individual subapplications. The algorithm of Figure 4 is chosen as an example. In Figure 25a, the individual subgraphs are transferred to the VPU consecutively, with either B_1 or B_2 being loaded. In Figure 25b, all subgraphs are transferred to a number of VPUs and
30   connected to one another via bus systems. Thus large amounts of data can be processed efficiently without the negative effect of the reconfiguration.

Figure 26 shows a circuit for speeding up the
35   (re)configuration time of PAEs. At the same time, the circuit can be used for processing sequential algorithms. The array of PAEs (2605) is partitioned into a plurality of portions (2603). An independent unit for (re)configuration (2602) is

associated with each portion. A CT (2601) according to the related art (see PACT10) is at a higher level than these units and is in turn connected to another CT or a memory (2604). The CT loads the algorithms into the configuration units (2602). The 2602 automatically load the configuration data into the PAEs associated with them.

Figure 27 shows the structure of a configuration unit. The core of the unit is a sequencer (2701), which has a series of commands.
The most important commands are:

wait <trg#>
Wait for the receipt of a certain trigger from the array, which indicates which next configuration should be loaded.

lookup <trg#>
Returns the address of the subprogram called by a trigger received.

jmp <adr>
Jump to address

call <adr>
Jump to address. Return jump address is stored on the stack.

jmp <cond><adr>
Conditional jump to address

call <cond><adr>
Conditional jump to address. Return jump address is stored on the stack.

ret
Return jump to the return jump address stored on the stack

mov <target> <source>
Transfers a data word from source to target. Source and target

may each be a peripheral address or in a memory.

The commands are basically known from PACT10, i.e., the description of the CT. The basic difference is in the implementation of 2602, that only very simple commands are used for data management and no complete microcontroller is used.

One significant extension of the command set is the "pabm" command for configuring the PAEs. Two commands (pabmr, pabmm) are available, which have the following structure:

a)

pabmr	regno	count
pa_adr_0		pa_dta_0
pa_adr_1		pa_dta_1
...		...
pa_adr_{count}		pa_dta_{count}

pabmr	0	count
offset		
pa_adr_0		pa_dta_0
pa_adr_1		pa_dta_1
...		...
pa_adr_{count}		pa_dta_{count}

b)

pabmr	regno	count
memref		

pabmm	0	count
offset		
memref		

The commands copy an associated block of PAE addresses and PAE

data from the memory to the PAE array. <count> indicates the
size of the data block to be copied. The data block is either
directly appended to the opcode (a) or referenced by
specifying the first memory address <memref> (b).

5

Each pa_adr_n-pa_dta_n row represents a configuration for a PAE.
pa_adr_n specifies the address and pa_dta_n specifies the
configuration word of the PAE.

10   The RDY/ACK-REJ protocol is known from PACT10. If the
configuration data is accepted by a PAE, the PAE acknowledges
the transmitted data with an ACK. However, if a PAE cannot
accept the configuration data because it is not in a
reconfigurable state, it returns a REJ. Thus the configuration
15   of the subalgorithm fails.

The location of the pa_adr_n-pa_dta_n row rejected with REJ is
stored. The commands are called again at a later time (see
PACT10, FILMO). If the command was completely executed, i.e.,
20   no REJ occurred, the command performs no further
configuration, but terminates immediately. If a REJ occurred,
the command jumps directly to the location of the rejected
pa_adr_n-pa_dta_n row. Depending on the command, the location is
stored in different ways:
25   pabmr: the address is stored in the register named <regno>;
pabmm: the address is stored directly in the command at the
memory location <offset>.

The commands can be implemented via DMA structures as
30   memory/IO transfers according to the related art. The DMAs are
extended by a logic for monitoring the incoming ACK/REJ. The
start address is determined by <regno> or <offset>. The last
address of the data block is computed via the address of the
command plus its opcode length minus one plus the number of
35   pa_adr_n-pa_dta_n rows.

It is also useful to extend the circuit described in PACT10 by
the above-mentioned commands.

Figure 27 shows the structure of a 2602 unit. The unit has a register set 2701 with which a simple ALU is associated for stack operations (2702). The structure contains address registers and stack pointers. Optionally, a full-fledged ALU

5   can be used. A bus system (2703) having a minimum width connects registers and ALU. The width is such that simple control flow commands or simple ALU operations can be represented practically. The above-described PABM commands and the commands according to PACT10 are also supported. Registers

10 and ALU are controlled by a sequencer 2706, which represents a complete microcontroller by its execution of commands.

A unit 2704, which receives and acknowledges triggers from the associated PAEs and transmits triggers to the PAEs when

15 appropriate, is connected to 2703. Incoming triggers cause an interrupt in sequencer 2706 or are queried by the WAIT command. Optionally, an interface (2705) to a data bus of the associated PAEs is connected to 2703 in order to be able to send data to the PAEs. For example, the assembler codes of a

20 sequencer implemented in the PAEs are transmitted via 2705. The interface contains, when required, a converter for adjusting the different bus widths. Units 2701 through 2706 are connected to a bus system (2708), which is multiple times wider and leads to the memory (2709), via a

25 multiplexer/demultiplexer (2707). 2707 is activated by the lower-value addresses of the address/stack register; the higher-value addresses lead directly to the RAM (2711). Bus system 2708 leads to an interface (2709), which is controlled by the PA commands and leads to the configuration bus of the

30 PAEs. 2708 is designed to be wide enough to be able to send as many configuration bits as possible per cycle unit to the PAEs via 2709. An additional interface (2710) connects the bus to a higher-level CT, which exchanges configuration data and control data with 2602. Interfaces 2710 and 2709 have been

35 repeatedly described in PACT10, PACT??.

It is essential that 2706 has a reduced, minimum set of commands that is optimized for the task, mainly for PA

commands, jumps, interrupts, and lookup commands. Furthermore, optimized wide bus system 2708, which is transferred to a narrow bus system via 2707 is of particular importance for the reconfiguration speed of the unit.

Figure 27a is a special version of Figure 27. Interface 2705 is used for transmitting assembler codes to sequencers configured in the PAE array. The processing capacity of the sequencers essentially depends on the speed of interface 2705 and of its memory access. In Figure 27a, 2705 is replaced by a DMA function with direct memory access (2720_n). 2720_n performs its own memory accesses and has its own bus system (2722_n) with appropriate adjustment of the bus width (2721_n); the bus can be relatively wide for loading wide command sequences (ULIW), so that in the limit case 2721_n is not needed at all. In order to further increase the speed, memory 2711 has been physically separated into 2711a and $2711b_n$. The address space across 2711a and $2711b_n$ remains linear, but 2701, 2701, and 2706 can access both memory blocks independently and simultaneously; 2720_n can only access $2711b_n$. 2720_n, 2721_n, and $2711b_n$ can be implemented as multiple units $(_n)$, so that more than one sequencer can be managed at the same time. For this purpose, $2711b_n$ can be subdivided again into multiple physically independent memory areas. Examples of implementation for 2720_n are described in Figure 38.

Figure 28 illustrates the structure of complex programs. The basic modules of the programs are the complex configurations (2801) containing the configurations of one or more PAEs and the respective bus and trigger configurations. 2801 are represented by an opcode (2802), which may have additional parameters (2803). These parameters may have constant data values, variable start values or even special configurations. Depending on the function, there is one parameter, a plurality of parameters, or no parameter.

Multiple opcodes use a common set of complex configurations to form an opcode group (2805). The different opcodes of a group

differ from one another by the special versions of the complex
configurations. Differentiation elements (2807) which either
contain additional configuration words or overwrite
configuration words occurring in 2801 are used for this
5   purpose.

If no differentiation is required, a complex configuration is
called directly by an opcode (2806). A program (2804) is
composed of a sequence of opcodes having the respective
10 parameters.

A complex function can be loaded once into the array and then
reconfigured again by different parameters or
differentiations. Only the variable portions of the
15 configuration are reconfigured. Different opcode groups use
different complex configurations. (2805a, ..., 2805n).

The different levels (complex configuration, differentiation,
opcode, program) are run in different levels of CTs (see CT
20 hierarchies in PACT10). The different levels are illustrated
in 2810, with 1 representing the lowest level and N the
highest. CTs with hierarchies of any desired depth can be
constructed (see PACT10).

25 A distinction is made in 2801 between two types of code:

1. Configuration words which map an algorithm onto the array
of PAEs. The algorithm can be designed as a sequencer.
Configuration takes place via interface 2709. Configuration
30 words are defined by the hardware.

2. Algorithm-specific codes, which depend on the possible
configuration of a sequencer or an algorithm. These codes are
defined by the programmer or the compiler and are used to
35 activate an algorithm. If, for example, a Z80 is configured as
a sequencer in the PAEs, these codes represent the opcode of
the Z80 microprocessor. Algorithm-specific codes are
transmitted to the array of PAEs via 2705.

Figure 29 shows a possible basic structure of a PAE. 2901 and 2902 represent, respectively, the input and output registers of the data. The complete interconnection logic to be connected to the data bus(es) (2920, 2921) of the array is associated with the registers (see PACT02). The trigger lines according to PACT08 are tapped from the trigger bus (2922) by 2903 and connected to the trigger bus (2923) via 2904. An ALU (2905) of any desired configuration is connected between 2901 and 2902. A register set (2915) in which local data is stored is associated with the data buses (2906, 2907) and with the ALU. The RDY/ACK synchronization signals of the data buses and trigger buses are supplied (2908) to a state machine (or a sequencer) (2910) or generated by the unit (2909).

The CT selectively accesses a plurality of configuration registers (2913) via an interface unit (2911) using a bus system (2912). 2910 selects a certain configuration via a multiplexer (2914) or sequences via a plurality of configuration words which then represent commands for the sequencer.

Since the VPU technology operates mainly pipelined, it is of advantage to additionally provide either groups 2901 and 2903 or groups 2902 and 2904 or both groups with FIFOs. This can prevent pipelines from being jammed by simple delays (e.g., in the synchronization).

2920 is an optional bus access via which one of the memories of a CT (see Fig. 27, 2720) or a conventional internal memory can be connected to sequencer 2910 instead of the configuration registers. This allows large sequential programs to be executed in one PAE. Multiplexer 2914 is switched so that it only connects the internal memory.

The addresses are

a) generated for the CT memory by the circuit of Fig. 38;

b) generated directly by 2910 for the internal memory.

Figure 30 shows a possible extension of the PAE in order to allow the CT or another connected microprocessor to access the data registers. The address space and the interface of the bus unit (formerly 2911, 3003) are extended by the additional data buses (3001). A multiplexer (3002), through which 3003 can write data into the register via bus 3001, is connected upstream from each register. The outputs of the registers are looped back to 3003 via 3001. 3003 transmits the data to CT 2912. As an alternative (3003a), the data can be transmitted to a bus (3005) that is independent of CT via an additional interface (3004) in order to transmit the data to CT.

Figure 31 shows the connection of the array of PAEs (3101) to a higher-level microcontroller. 3101 contains all IO channels according to the memories according to the present invention. The architecture operates as shown in Fig. 23. 2912 in Figure 31a provides the bus for the configuration data and register data according to Figure 30. The data bus is shown separately by 3104. 3102 represents the CT, which in Figure 31a also represents the microprocessor.

For all bus systems, there are the following connection models to a processor which are selected depending on the programming model and balancing price and performance.

1. Register model
In the register model, the respective bus is addressed via a register, which is directly integrated in the register set of the processor and is addressed by the assembler as a register or a group of registers. This model is most efficient when a few registers suffice for the data exchange.

2. IO model
The respective bus is located in the IO area of the processor. This is usually the simplest and most cost-effective version.

3.   Shared memory model

Processor and respective bus share one memory area in the data
memory storage device. This is an effective version for large
amounts of data.

5

4.   Shared memory-DMA model

Processor and bus share the same memory as in the previous
model. There is a fast DMA to further increase speed (see
Figure 38), which takes on the data exchange between bus and

10   memory.

In order to increase the transmission speed, the respective
memories should be physically separable from the other
memories (a plurality of memory banks), so that processor and

15   VPU can access their memories independently.

In Figure 31b, a CT (3102) performs the configuration of the
array, while a dedicated processor (3103) guarantees the
programming model according to Fig. 23 via 3006 by exchanging

20   register data with the array via 3006 and exchanging
conventional data via 3104.

Figures 31c/d correspond to Figures 31a/b, but a shared memory
(3105) is selected for data exchange between the respective

25   processor and 3101.

Figure 32 shows a circuit which allows the memory elements
according to the present invention to jointly access a memory
or a group of memories; each individual memory of the group

30   can be individually and uniquely addressed. For this purpose,
the individual memory elements (3201) are connected to a bus
system, in which each 3201 has its own bus. The bus can be
bidirectional or implemented by two unidirectional buses.
There is an address/data multiplexer for each memory, which

35   connects a bus to the memory. For this purpose, the adjacent
addresses of each bus are decoded (3207) and then one bus per
time unit is selected (3204) by an arbiter (3208). The
corresponding data and addresses are transferred to the

respective memory bus (3205a), with a state machine (3206) generating the required protocols. If the data are received from the memory upon a read request, the respective state machine sends the address of the memory to the bus that

5   requested the data. The addresses of all incoming buses are evaluated by a multiplexer unit for each bus of the bus system (3202) and transferred to the respective bus. The evaluation takes place corresponding to the evaluation of the output data, i.e., a decoder (3209) for each input bus (3205b)

10   conducts a signal to an arbiter (3210) which activates the data multiplexer. Thus different input buses are connected to the bus system (3202) in each time unit.

In Figure 33, the rigid state machine / rigid sequencer 2910 is replaced by a freely programmable one (3301) for a simpler

15   and more flexible evaluation of the trigger and RDY/ACK signals. The full function of 3301 is determined by the configuration registers (2913) prior to the execution of algorithms by the CT. Loading of 3301 is controlled by a CT

20   interface (3302) which has been extended by the management of 3301 with respect to 2911. The advantage of 3301 is that it allows handling of the different trigger and RDY/ACK signals in a much more flexible manner than in fixedly implemented 2910. The disadvantage is the size of a 3301.

25

A compromise resulting in maximum flexibility and a reasonable size is evaluating the trigger and RDY/ACK signals by a unit according to 3301 and controlling all fixed processes within the PAE by a fixedly implemented unit according to 2910.

30

The PAE according to the present invention for processing logical functions is illustrated in Figure 34. The core of the unit is a unit described in detail below for gating individual signals (3401). The bus signals are connected to 3401 via the

35   known registers 2901, 2902, 2903, 2904. The registers are extended by a feed mode for this purpose, which selectively exchanges individual signals between the buses and 3401 without storing them (register) in the same cycle. The

multiplexer (3402) and the configuration registers (3403) are adjusted to the different configurations of 3401. The CT interface (3404) is also configured accordingly.

5   Figure 35 shows possible embodiments of 3401. A global data bus connects logic cells 3501 and 3502 to registers 2901, 2902, 2903, 2904. 3504 is connected to the logic cells via bus switches, which can be designed as multiplexers, gates, transmission gates, or simple transistors. The logic cells may
10   be designed to be completely identical or may have different functionalities (3501, 3502). 3503 represents a RAM.

Possible designs of the logic cells include:
- lookup tables,
15   - logic
- multiplexers
- registers

The selection of the functions and interconnection can be
20   either flexibly programmable via SRAM cells or using read-only ROMs or semistatic Flash ROMs.

In order to speed up sequential algorithms, which are difficult to parallelize, speculative design is related art
25   with conventional processors. The parallel version for VPUs is shown in Figure 36. The operands (3601) go to a plurality of possible paths of subalgorithms (3602a, 3602b, 3602c) at the same time. The subalgorithms may have different area and time requirements. Depending on the subalgorithms, the data is
30   stored according to the present invention (3612a, 3612b, 3612c) before being processed (3603) by the next subalgorithms after reconfiguration. The times of reconfiguration of the individual subalgorithms are also independent of one another, as is the number of subalgorithms themselves (3603, 3614). As
35   soon as it can be decided which of the paths is to be selected, the paths are combined via a bus or a multiplexer (3605). Trigger signals generated by a condition (see PACT08) (3606) determine which of the paths is selected and forwarded

to the next algorithms.

Figure 37 shows the design of a high-level language compiler,
which translates common sequential high-level languages (C,
Pascal, Java) to a VPU system. Sequential code (3711) is
separated from parallel code (3708), whereby 3708 is directly
processed in the array of PAEs.

There are three design options for 3711:

1. Within a sequencer of a PAE (2910).
2. Via a sequencer configured in the VPU. To do so, the
compiler generates a sequencer optimized for the task, as well
as the algorithm-specific sequencer code (see 2801) directly.
3. On a conventional external processor (3103).

The option selected depends on the architecture of the VPU, of
the computer system, and of the algorithm.

The code (3701) is initially separated in a pre-processor
(3702) into data flow code (3716) (written in a special
version of the respective programming language and optimized
for the data flow), and common sequential code (3717). 3717 is
checked for parallelizable subalgorithms (3703), and the
sequential subalgorithms are eliminated (3718). The
parallelizable subalgorithms are placed temporarily as macros
and routed.

In an iterative process, the macros are placed together with
the data flow-optimized code (3713), routed, and partitioned
(3705). A statistical unit (3706) evaluates the individual
macros and their partitioning with regard efficiency, with the
time and the resources used for reconfiguration being factored
into the efficiency evaluation. Inefficient macros are removed
and separated as sequential code (3714).

The remaining parallel code (3715) is compiled and assembled
(3707) together with 3716, and VPU object code is output

(3708).

Statistics concerning the efficiency of the code generated and
of the individual macros (including those removed with 3714)
5   are output (3709); thus the programmer receives essential
information on the speed optimization of the program.

Each macro of the remaining sequential code is checked for
complexity and requirements (3720). The appropriate sequencer
10   is selected from a database, which depends on the VPU
architecture and the computer system (3719), and output as VPU
code (3721). A compiler (3721) generates and outputs (3711)
the assembler code of the respective macro for the sequencer
selected by 3720. 3710 and 3720 are closely linked. The
15   process may take place iteratively in order to find the most
suitable sequencer with the least and fastest assembler code.

A linker (3722) combines the assembler codes (3708, 3711,
3721) and generates the executable object code (3723).
20

Figure 38 shows the internal structure of 2720. The core of
the circuit is a loadable up/down counter (3801), which gets
its start value from bus 3803 (corresponds to 2703) of the
circuit of Fig. 27 via appropriately set multiplexer 3802. The
25   counter is used as a program counter (PC) for the associated
sequencer; the start value is the first address of the program
to be executed. The value of 3801 is looped back to the
counter via an adder (3805) and 3802. An offset, which is
either subtracted from or added to the PC, is sent by the
30   sequencer to 3805 via bus 3804. Thus relative jumps can be
efficiently implemented. The PC is supplied to the PAE array
via bus 3811 and can be stored on the stack for call
operations. For ret operations, the PC is sent from the stack
to 3801 via 3804 and 3802.
35

Either the PC or a stack pointer (3807) supplied by the PAE
array is supplied to an adder (3808) via multiplexer 3806.
Here an offset which is stored in register 3809 and written

via 3803 is subtracted from or added to the values. 3808 allows the program to be shifted within memory 2711. This enables garbage collector functions to clean up the memory (see PACT10). The address shift which occurs due to the garbage collector is compensated for by adjustment of the offset in 3809.

Figure 38a is a variant of Figure 38 in which the stack pointer (3820) is also integrated. Only the offset is supplied to 3805 via 3804 for relative jumps (3804a). The stack pointer is an up/down counter similar to 3801, whose start value represents the beginning of the stack and is loaded via 3803. The PC is sent directly to the data bus for the memory in order to be written onto the stack via a multiplexer in the event of call operations. The data bus of the memory is looped back to 3801 via 3821 and 3802 to perform ret operations.

Figure 39 illustrates the mode of operation of the memories. The memory (3901) is addressed via a multiplexer (3902). In the standard mode, lookup mode, and register mode, the addresses are supplied from the array (3903) directly to 3901. In the stack mode and FIFO mode, the addresses are generated in an up/down counter (3904). In this case, the addresses are supplied to the IO side by another up/down counter (3905). The addresses for the external RAM (or IO) are generated by another up/down counter (3906); the base address is loaded from a register (3907). The register is set by the CT or an external host processor. A state machine (3908) takes over the entire control. 3908 reads the status of the memory (full, empty, half-full, etc.) in an up/down counter (3909), which counts the number of words in the memory. If the memory is modified block by block (write stack onto external stack or read from external stack), the size of the block is supplied as a constant (3917) to an adder/subtracter (3910), to which the count of 3909 is looped back. The result is loaded according to 3909.

Thus the count can be rapidly adjusted to block-by-block

changes. (Of course, it is also possible to modify the counter with each written or read word in a block operation.) For cache operations, a cache controller (3911) according to the related art is available, which is associated with a tag
5   memory (3912). Depending on the mode of operation, the value of 3911 or 3906 is sent out (3914) via a multiplexer (3913) as an address. The data is sent out via bus 3915, and data is exchanged with the array via bus 3916.

10   **Programming examples to illustrate the subalgorithms**
A module can be declared in the following way, for example:

    module example1
    input (var1, var2 : ty_1; var3 : ty_2).
15   *output* (res1, res2 : ty_3).
    begin
    . . .
    register <regname1> (res1).
    register <regname2> (res2).
20   *terminate@* (res1 & res2; 1).
    end.

module identifies the beginning of a module.
input/output defines the input/output variables with the types
25   ty_n.
begin ... end mark the body of the module.
register <regname1/2> transfers the result to the output, the result being temporarily stored in the register specified by <regname1/2>. <regname1/2> is a global reference to a certain
30   register.

The following memory types are available, for example, as additional transfer modes to the output:

35   *fifo* <fifoname>, where the data is transmitted to a memory operating by the FIFO principle. <fifoname> is a global reference to a specific memory operating by the FIFO

principle. **terminated@** is extended by the "fifofull"

parameter, i.e., signal, which shows that the memory is full.

stack <stackname>, where the data is transmitted to a memory

operating by the stack principle. <stackname> is a global

5 reference to a specific memory operating in the stack mode.

terminate@ differentiates the programming by the method

according to the present invention from conventional

sequential programming. The command defines the abort

10 criterion of the module. The result variables res1 and res2

are not evaluated by terminate@ with their actual values, but

only the validity of the variables (i.e., their status signal)

is checked. For this purpose, the two signals res1 and res2

are gated with one another logically via an AND, OR, or XOR

15 operation. If both variables are valid, the module is

terminated with the value 1. This means that a signal having

value 1 is forwarded to the higher-level load unit, whereupon

the higher-level load unit loads the next module.

20
```
module example2
input (var1, var2 : ty₃; var3 : ty₂).
output (res1 : ty₄).
begin
register <regname1> (var1, var2).
```
25
```
...
fifo <fifoname1> (res1, 256).
terminate@ (fifofull(<fifoname1>); 1).
end.
```

30 **register** is defined via input data in this example. <regname1>

is the same here as in example1. This causes the register,

which receives the output data in example1, to provide the

input data for example2.

35 **fifo** defines a FIFO memory with a depth of 256 for the output

data res1. The full flag (fifofull) of the FIFO memory is used

as an abort criterion in **terminate@**.

```
        module main
        input (in1, in2 : ty₁; in3 : ty₂).
        output (out1 : ty₄).
        begin
5       define <regname1> : register(234).
        define <regname2> : register(26).
        define <fifoname1> : fifo(256,4).   //FIFO depth 256
        ...
        (var12, var72) = call example1 (in1, in2, in3).
10      ...
        (out1) = call example2 (var12, var72, var243).
        ...
        signal (out1).
        terminate@ (example2).
15      end.
```

define defines an interface for data (register, memory, etc.).
The required resources and the name of the interface are
specified with the definition. Since each of the resources is
20 only available once, they must be specified unambiguously.
Thus the definition is global, i.e., the name is valid for the
entire program.

call calls a module as a subprogram.

25

signal defines a signal as an output signal without a buffer
being used.

The module main is terminated by **terminate@** (example2) as soon
30 as subprogram example2 is terminated.

In principle, due to the global declaration "define..." the
input/output signals thus defined do not need to be included
in the interface declaration of the modules.

35

What is claimed is:

1. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein the data flow graph and the control flow graph of a program is extracted.

2. The method according to Claim 1,
wherein the graphs are broken down into a plurality of subgraphs so as to obtain as few connections as possible between the subgraphs.

3. The method according to Claim 1,
wherein the graphs are broken down into a plurality of subgraphs so that as little data as possible is transmitted between the subgraphs.

4. The method according to Claim 1,
wherein the charts are broken down into a plurality of subcharts so that as far as possible no loop-back is obtained between the subgraphs.

5. The method according to Claim 1,
wherein the graphs are broken down into a plurality of subgraphs so that the subgraphs are adjusted to the resources of the module as exactly as possible.

6. The method according to Claim 1,
wherein memory elements are inserted between the subgraphs for saving the data and states.

7. The method according to Claim 1,
wherein status signals are transmitted between the nodes within a subgraph making the state of each individual node available to each of the other nodes.

8. The method according to Claims 1 and 7,
wherein a number of status signals is sent to a higher-level

unit which controls the configuration of the cells in order to trigger reconfiguration.

9. The method according to Claims 1 and 7,
wherein a number of status signals is sent to a higher-level unit which controls the configuration of the cells in order to transmit a status to a subgraph which is not loaded into the cell structure.

10. The method according to Claim 1,
wherein the subgraphs are distributed among a plurality of modules.

11. The method according to Claim 1,
wherein a plurality of paths of an instruction, one of which is always executed depending on the evaluation of the instruction (IF, CASE), is broken down so that each path yields a subgraph.

12. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein a status which indicates whether or not the signal is valid (RDY) is associated with each data signal.

13. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein a status which indicates whether or not the signal is valid (RDY) is associated with each status signal.

14. The method according to Claims 12 and 13,
wherein the receiver of a valid signal acknowledges receipt (ACK).

15. The method according to Claims 12 and 13,
wherein the receiver indicates that it expects a signal (REQ).

16. The method according to Claim 15,
wherein the transmitter indicates that it is transmitting the

expected signal.

17.   A method of programming modules having a unidimensional
or multidimensional cell structure,
wherein a first part of the cell structure computes a subgraph
and terminates its computation step by step; as soon as one or
more cells have terminated the computation, they are
reconfigured as the second part of the cell structure, so that
a third part **[computes]** the new subgraph simultaneously with
the reconfigured cells (wa**[v]**e reconfig).

18.   The method according to Claim 17,
wherein a plurality of configuration registers of a cell store
different configurations of different subgraphs at the same
time.

19.   The method according to Claims 17 and 18,
wherein one configuration is active out of a plurality of
configurations.

20.   The method according to Claims 17 and 18, wherein
unconfigured configuration registers are specially marked.

21.   The method according to Claim 17,
wherein the respective configuration is selected by status
signals generated by the cell structure.

22.   The method according to Claim 17,
wherein the respective configuration is selected by status
signals generated by a higher-level load unit.

23.   The method according to Claim 17,
wherein the respective configuration is selected by externally
generated status signals.

24.   The method according to Claims 17 and 21 through 23,
wherein each cell evaluates the status signals individually
according to its configuration and activates the respective

configuration.

25. The method according to Claims 17 and 20 through 24,
wherein, when an unconfigured configuration register is
activated, the configuration is requested from the higher-
level load unit and the execution of the subgraphs is
suspended until the configuration is fully loaded.

26. The method according to Claim 17,
wherein loading of a configuration is triggered by status
signals generated by the cell structure.

27. The method according to Claim 17,
wherein loading of a configuration is triggered by a higher-
level load unit.

28. The method according to Claim 17,
wherein loading of a configuration is triggered by externally
generated status signals.

29. The method according to Claims 17 and 26 through 28,
wherein each cell evaluates the status signals individually
according to its configuration and causes the respective
configuration to be loaded.

30. A method of programming modules having a unidimensional
or multidimensional cell structure,
wherein sequencers, which address the configuration registers
and execute a program stored in the configuration registers,
are integrated in the cells.

31. A method of programming modules having a unidimensional
or multidimensional cell structure,
wherein sequencers, which address a memory associated with the
cell structure and execute a program stored in the memory, are
integrated in the cells.

32. A method of programming modules having a unidimensional

or multidimensional cell structure,
wherein a sequencer configured according to the program to be
executed is formed by interconnecting a plurality of cells.

33. A method of programming modules having a unidimensional
or multidimensional cell structure,
wherein an external processor is coupled to the module in
order to execute sequential subgraphs.

34. The method according to Claim 33,
wherein the higher-level load unit is also used as a processor
for executing sequential subgraphs.

35. The method according to Claims 30 through 34,
wherein the sequencers have access to the data registers of
the individual cells.

35. The method according to Claims 30 through 34,
wherein the remaining subgraphs are configured to the
sequencer according to their respective design.

36. The method according to Claims 30 through 35,
wherein the remaining subgraphs are reconfigured to the
sequencer according to their respective design.

37. The method according to Claims 30 through 34,
wherein standard arithmetic units corresponding to the CISC
model are configured to the sequencer.

38. The method according to Claims 30 through 34 and 37,
wherein appropriate commands to activate the standard
arithmetic units are generated by the compiler and a plurality
of subgraphs are mapped onto a standard arithmetic unit.

39. The method according to Claims 30 through 34 and 37
through 38,
wherein appropriate commands for the external connection of
the standard arithmetic units are generated by the compiler

and a plurality of subgraphs are mapped onto a standard
arithmetic unit.

40.   The method according to Claims 30 through 34 and 37
through 38,
wherein appropriate commands for the internal connection of
the standard arithmetic units are generated by the compiler
and a plurality of subgraphs are mapped onto a standard
arithmetic unit.

41.   The method according to Claims 30 through 34 and 37
through 40,
wherein the commands are loaded cyclically as determined by a
program counter.

42.   The method according to Claims 30 through 34,
wherein the sequencer manages its operands on a stack and
represents a stack processor.

43.   The method according to Claims 30 through 34,
wherein the sequencer manages its operands in an accumulator
and represents an accumulator processor.

44.   The method according to Claims 30 through 34,
wherein the sequencer manages its operands in a register set
and represents a register processor.

45.   The method according to Claims 30 through 34,
wherein the sequencer manages its operands in a memory and
represents a load/store processor.

46.   The method according to Claims 30 through 34 and 42
through 45,
wherein the sequencer has simultaneously implemented different
procedures suitable for executing the program.

47.   The method according to Claims 30 through 34 and 42
through 46,

wherein a plurality of sequencers of different designs are configured simultaneously in the cell structure.

48. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein pointers occurring in the program are re-sorted so that they have the greatest possible time independence, i.e., as many commands not depending on a pointer as possible are located between two pointers.

49. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein pointers occurring in the program are re-sorted so that the data referenced by the pointer is used downstream as far as possible from the calculation of the pointer.

50. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein, in the event of jumps and comparisons, all possible subgraphs are configured and computed simultaneously in the cell structure until it is known which subgraph is selected by the jump or comparison.

51. The method according to Claim 50,
wherein the data and states of all unselected subgraphs are ignored, and only the data and states of the selected subgraphs are further processed.

52. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein one or more memories are associated with the cell structure.

53. The method according to Claim 52,
wherein the memory is addressed as freely as desired (random access).

54. The method according to Claim 52,

wherein the memory is used as a lookup table.

55. The method according to Claim 52,
wherein the memory is used as a FIFO memory for isolating data
streams.

56. The method according to Claim 52,
wherein the memory is used as a stack for a sequencer.

57. The method according to Claim 52,
wherein the memory is used as a register bank for a sequencer.

58. The method according to Claims 52 through 54 and 56
through 57,
wherein the memory represents a section of the external
memory.

59. The method according to Claims 52 through 53,
wherein the memory operates as a cache for the external
memory.

60. The method according to Claim 52,
wherein the memory is written into the external memory by a
signal from the cell structure.

61. The method according to Claim 52,
wherein the memory is written into the external memory by a
signal from the higher-level load unit.

62. The method according to Claim 52,
wherein the memory is read from the external memory by a
signal from the cell structure.

63. The method according to Claim 52,
wherein the memory is read from the external memory by a
signal from the higher-level load unit.

64. The method according to Claims 52 and 60 through 63,

66   **LITERAL TRANSLATION**

wherein the base address is set in the external memory free from the cell structure.

65. The method according to Claims 52 and 60 through 63, wherein the base address is set in the external memory free from the higher-level load unit.

66. The method according to Claims 52 and 60 through 63, wherein the base address is set in the external memory free from an external unit.

66. The method according to Claims 52, 57 and 60 through 66, wherein a switch-over from one subgraph to another (context switch with push/pop) is executed by the register bank writing into and reading from the external memory.

67. The method according to Claims 52 and 56, wherein the stack is larger than the memory in that parts of the stack are transferred onto the external memory.

68. A method of programming modules having a unidimensional or multidimensional cell structure, wherein the higher-level load units have a hierarchical structure.

69. The method according to Claim 68, wherein different parts of the configuration program are stored and/or run on each hierarchical level.

70. A method of programming modules having a unidimensional or multidimensional cell structure, wherein the higher-level load units have wide memories for the rapid transfer of the configuration data.

71. A method of programming modules having a unidimensional or multidimensional cell structure, wherein the memory width for the sequencers of higher-level load units is reduced via multiplexers.

72. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein the higher-level load units have a command for the transfer of the configuration data block by block.

73. The method according to Claim 72,
wherein the command for the transfer of configuration data block by block is implemented by the DMA principle.

74. The method according to Claim 72,
wherein the memory for the configuration data is accessed simultaneously and independently of the other memories.

75. The method according to Claim 72,
wherein a plurality of units exists for the transfer of the configuration data block by block.

76. The method according to Claims 72 and 75,
wherein independent accesses to the memories for the configuration data of the individual units take place simultaneously.

77. The method according to Claim 72,
wherein the command aborts when configuring a non-configurable cell.

78. The method according to Claims 72 and 77,
wherein the command stores the address of the configuration data of the non-configurable cell.

79. The method according to Claims 72 and 77 through 78,
wherein the command continues to operate in the event of a new execution at the location of the configuration data of the unconfigured cell.

80. The method according to Claims 72 and 77 through 79,
wherein the command is executed again only if a cell could not be configured.

81. A method of programming modules having a unidimensional or multidimensional cell structure, wherein the higher-level load units load configurations into their internal memories prior to these configurations being called.

82. The method according to Claim 81, wherein loading is executed through a command.

83. The method according to Claim 81, wherein loading is executed through a status signal.

84. A method of programming modules having a unidimensional or multidimensional cell structure, wherein configurations are combined to form groups.

85. The method according to Claim 84, wherein a group is variably customized by being called.

86. The method according to Claims 84 through 85, wherein the groups and their customization are stored on higher-level load units of a lower hierarchy.

87. The method according to Claims 84 through 86, wherein the calls of the groups are stored on higher-level load units of a higher hierarchy.

88. The method according to Claims 84 through 87, wherein programs are composed of a plurality of such calls.

89. A method of programming modules having a unidimensional or multidimensional cell structure, wherein configurations and command sequences of sequencers are stored in the higher-level load units.

90. A method of programming modules having a unidimensional or multidimensional cell structure, wherein command sequences of sequencers are contained in the

internal and/or external memories.

91. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein a distinction is made between the internal states of the sequencers and the states of the data processing.

92. The method according to Claim 91,
wherein the states of the data processing accompany the data in the cell structure.

93. The method according to Claims 91 through 92,
wherein the states of the data processing are saved with the data.

94. The method according to Claims 91 through 93,
wherein the states of the data processing are saved with each stored data word.

95. The method according to Claims 91 through 93,
wherein the states of the data processing are saved with the last stored data word before a reconfiguration.

96. A method of programming modules having a unidimensional or multidimensional cell structure,
wherein the address of the operand(s) last processed is saved before a reconfiguration.

97. The method according to Claim 96,
wherein the states of the data processing of the last operand are saved before a reconfiguration.

98. The method according to Claim 91,
wherein the internal states of the sequencers are not saved.

99. A method of compiling programs for modules having a unidimensional or multidimensional cell structure,
wherein a distinction is made between four types of code:

a) parallel code

b) code that can be efficiently parallelized

c) code that cannot be efficiently parallelized

d) sequential code.

100. The method according to Claim 99, wherein the parallel code is extracted from.

101. The method according to Claims 99 through 100, wherein the extracted code is placed and routed.

102. The method according to Claims 99 through 101, wherein partitioning is performed iteratively with placing and routing.

103. The method according to Claim 99, wherein the parallelizable code is extracted.

104. The method according to Claims 99 and 103, wherein the extracted code is placed and routed.

105. The method according to Claims 99 and 103 through 104, wherein partitioning is performed iteratively with placing and routing.

106. The method according to Claims 99 and 103 through 105, wherein each code is analyzed with regard to its efficiency and the codes that do not work efficiently are separated.

107. The method according to Claims 99 and 103 through 106, wherein statistics are prepared on which codes are efficient and which ones are inefficient, and appropriate instructions are given to the programmer for more efficient programming.

108. The method according to Claims 99 and 106, wherein the sequential and separated code is analyzed and an appropriate sequencer is selected for each individual code, a) a number of possible sequencers being predefined in a

database;

b) a single sequencer possibly being available (processor).

109. The method according to Claim 108,
wherein the code for the respective selected sequencer is
translated.

110. The method according to Claims 108 and 109,
wherein the selection of a sequencer and the translation take
place iteratively, in that each translation is analyzed with
regard to its efficiency, and the sequencer whose translated
code is most efficient is selected.

111. The method according to Claims 99 through 111,
wherein the translated codes and partitioned codes are
combined by a linker and the required communication structure
is inserted.

112. The method according to Claim 111,
wherein the linker inserts the required memories.

113. The method according to Claims 111 and 112,
wherein the linker inserts structures for saving the internal
states of the cell structure.

114. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein one or more memories are directly connected to the
cell structure.

115. The method according to Claim 114,
wherein the memory is accessed using word-by-word addressing.

116. The method according to Claim 114,
wherein the memory operates as a lookup table.

117. The method according to Claim 114,
wherein the memory operates as a stack for a sequencer.

118. The method according to Claim 114,
wherein the memory operates as a register set for a sequencer.

119. The method according to Claims 114 through 118,
wherein an interface to the peripheral device or external
memory is associated with or integrated into the memory.

120. The method according to Claims 114 through 119,
wherein the memory operates by the FIFO principle and thus it
isolates data streams in the cell structure from external data
streams.

121. The method according to Claims 114 and 119,
wherein the memory operates as a cache between the cell
structure and the external memory.

122. The method according to Claims 114 through 121,
wherein the memory stores code for a sequencer implemented in
the cell array.

122. The method according to Claim 114,
wherein a plurality of memories access a common peripheral
bus.

123. The method according to Claims 114 and 122,
wherein an arbiter always selects precisely one memory per
common bus and connects it to the bus via a multiplexer.

124. The method according to Claim 114,
wherein a status signal causes the contents of the memory to
be written onto the external memory.

125. The method according to Claim 114,
wherein a status signal causes the contents of the memory to
be read from the external memory.

126. The method according to Claims 114 and 124 through 125,
wherein the base address of the external memory is stored in a

register.

127. The method according to Claims 114 and 124 through 126, wherein the register is set by the cell structure.

128. The method according to Claims 114 and 124 through 126, wherein the register is set by the higher-level load unit.

129. The method according to Claims 114 and 124 through 126, wherein the register is set by the peripheral device.

130. The method according to Claim 117,
wherein the stack has a variable size in that the external memory is used to enlarge the stack.

131. The method according to Claims 117 and 130,
wherein the oldest part of the stack is written onto the stack in the external memory before a stack overrun.

132. The method according to Claims 117 and 130,
wherein the most recent part of the stack is read from the stack in the external memory before a stack underrun.

133. The method according to Claim 114,
wherein the memory provides its status via status signals.

134. The method according to Claims 114 and 121,
wherein a TAG memory is associated with the memory for the cache function.

135. The method according to Claims 114 and 119,
wherein the interface to the peripheral device is synchronized with the interface to the cell structure.

136. The method according to Claims 114 and 119,
wherein a unit for monitoring the addresses is associated with the interface to the peripheral device.

137. The method according to Claims 114 and 119,
wherein a unit for translating the addresses is associated
with the interface to the peripheral device.

138. The method according to Claim 114,
wherein the memory is designed as a ring memory.

139. The method according to Claims 114 and 119 and 138,
wherein the cell structure and the peripheral device each have
a position pointer.

140. The method according to Claims 114 and 138,
wherein a register provides the number of records in the
memory.

141. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein the status of a cell is forwarded to any other cells.

142. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein a cell has a plurality of configuration registers.

143. The method according to Claim 142,
wherein one of the configuration registers is selected at
runtime.

144. The method according to Claims 142 through 143,
wherein the selection takes place via a status signal within
the cell structure.

145. The method according to Claims 142 through 143,
wherein the selection takes place via a status signal of the
cell.

146. The method according to Claims 142 through 143,
wherein the selection takes place via a signal from the
higher-level load unit.

147. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein a sequencer is integrated in the cell.

148. The method according to Claims 142 and 147,
wherein a sequencer implemented in the cell selects the configuration register.

149. The method according to Claims 142 and 147,
wherein the sequencer analyzes the configuration word as a command.

150. The method according to Claim 147,
wherein the sequencer responds to the status signals of the cell structure.

151. The method according to Claim 147,
wherein the sequencer responds to the status signals of the cell.

152. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein a cell operates as an accumulator processor.

153. The method according to Claim 152,
wherein an accumulator is integrated in the cell.

154. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein a cell operates as a register processor.

155. The method according to Claim 154,
wherein a register set is integrated in the cell.

156. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein a cell operates as a stack processor.

157. The method according to Claim 156,
wherein the stack is integrated in a memory associated with
the cell structure.

158. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein a group of cells form an accumulator processor.

159. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein a group of cells form a stack processor.

160. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein a group of cells form a register processor.

161. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein a group of cells form a load/store processor.

162. The method according to Claims 158 through 161,
wherein a memory associated with the cell structure is
associated with the group.

163. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein the data registers of the cells are accessed by the
higher-level load unit.

164. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein the data registers of the cells are accessed by a
higher-level processor.

165. A method of operating modules having a unidimensional or
multidimensional cell structure,
wherein the data registers of the cells are accessed by
another cell configured as a processor.

166. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the data registers of the cells are accessed by other cells configured as processors.

167. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein a register set is implemented in a cell.

168. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the input registers of the cell are provided with FIFOs.

169. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the output registers of the cell are provided with FIFOs.

170. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the cell can be linked to a memory associated with the cell structure so that the codes for the sequencer implemented in the cell are loaded from the linked memory.

171. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the cell can be linked to the memory of a higher-level load unit so that the codes for the sequencer implemented in the cell are loaded from the memory of the higher-level load unit.

172. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the state machine of a cell is programmable.

173. A method of operating modules having a unidimensional or multidimensional cell structure,

wherein the state machine of a cell is partially programmable.

174. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the state machine of a cell is designed as a programmable logic.

175. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein the state machine and the arithmetic unit of a cell are designed as a programmable logic.

176. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein a unit for generating the addresses of the codes for access by a sequencer is provided for the higher-level load units in the cell structure.

177. A method of operating modules having a unidimensional or multidimensional cell structure,
wherein a unit for generating the address of a stack for access by a sequencer is provided for the higher-level load units in the cell structure.

178. The method according to Claims 176 through 177,
wherein address translation takes place by the records in the memory being shifted by an offset.

Abstract

The present invention is concerned with cell structures in which a changing arrangement with respect to one another is possible. How, and with the use of which units a code sequence is to be partitioned for this case and for this purpose, is mentioned.

LITERAL TRANSLATION

HDL-Code A → Synthesis → Binary Code A

HDL-Code B → Synthesis → Binary Code B

HDL-Code n → Synthesis → Binary Code n

Related Art

Fig. 1

Synthesis → **Conditionally Possible**

0201

for i= a to m
Synthesis →

1   2   3   n

0202

while
Synthesis → **Impossible**

0203
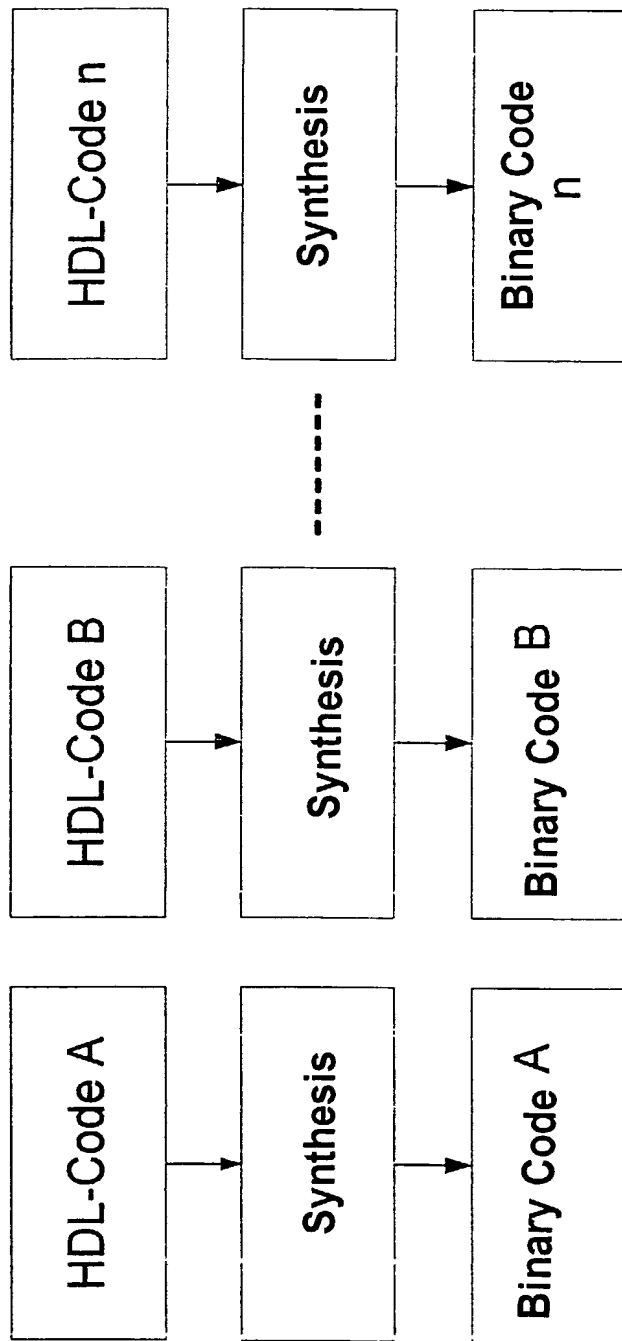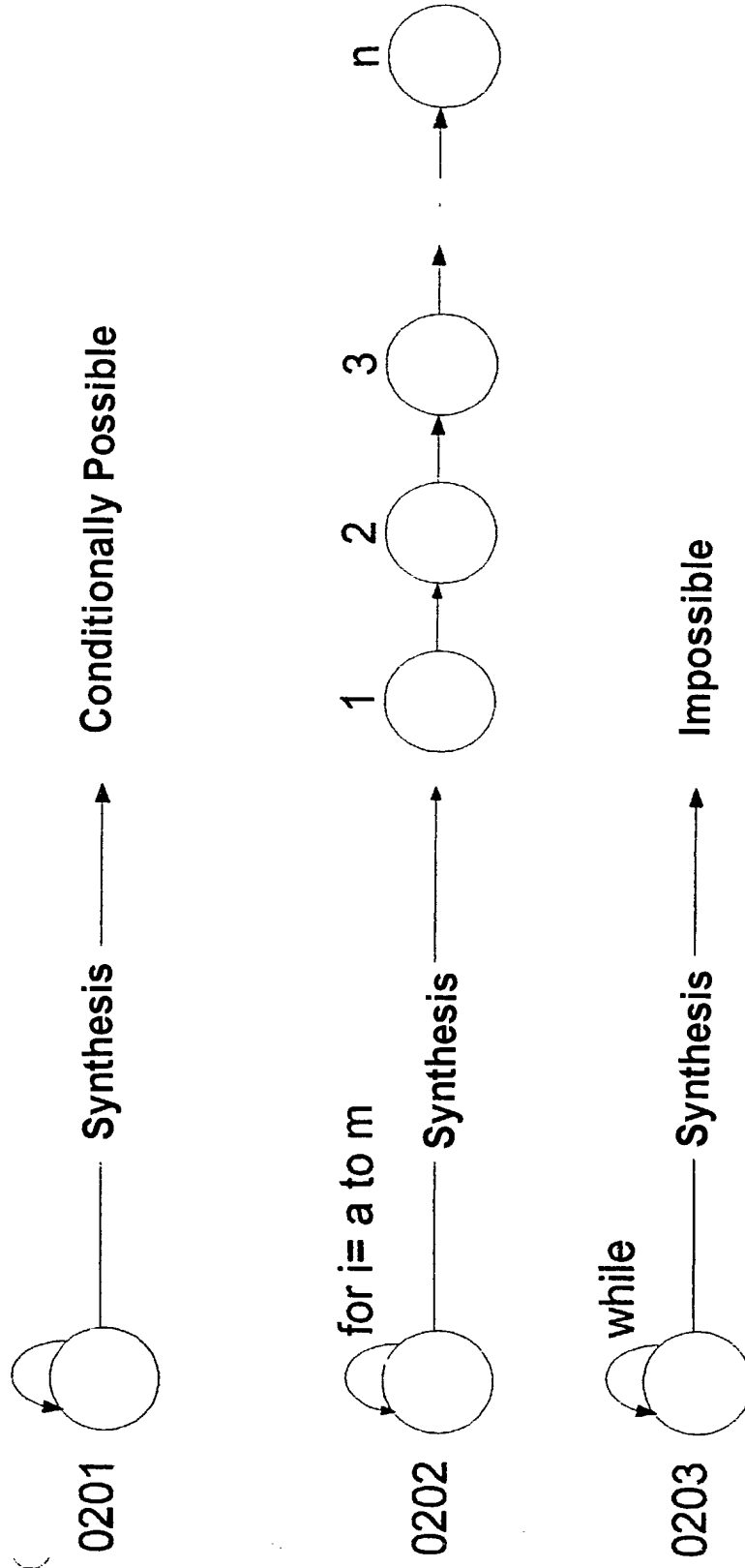
Fig. 2

Related Art

Fig. 3

Related Art

Fig. 4a

Fig. 4b

Fig. 5a

Fig. 5b

Fig. 6a

Fig. 6b

Fig. 7

Fig. 8b

Fig. 8a

Related Art

0904

CONFIG

Fig. 9a

ALU

STATUS

0903

PAE

Related Art

Fig. 9b

0902

DATA

0901

Fig. 10a

Fig. 10b

Fig. 10c

Fig. 11a

Fig. 11b

Fig. 12

Fig.

13a **13b** **13c** **13d** **13e**

PAE 9

PAE 8

PAE 7

PAE 6

PAE 5

PAE 4

PAE 3

PAE 2

PAE 1

Fig.

13f 13g 13h 13i 13j

PAE 9 | PAE 8 | PAE 7 | PAE 6 | PAE 5 | PAE 4 | PAE 3 | PAE 2 | PAE 1

Fig. 13g* 13h* 13i* 13j*

Fig. 14

ID H:
fetch A;   ← 1503
mul y, q, m;
mul x, q, n;
exec A;   ← 1504
exit;

ID A: ← 1506
fetch K;
sub c,a,b;   ← 1501
cmp c, 0;
settrg C, gt;
settrg K, lteq;   ← 1502
exit;

ID C:
sub d, x, c;
sub e, y, c;
exec B;

ID K:
add d, x, c;
add e, y, c;
exec B;   ← 1505

y := q * m
x := q * n

c := a - b

c > 0

d := x + c
e := y + c

d := x - c
e := y - c

Fig. 15

Fig. 16

1604

1602

1603

RDY
DATA
ACK

REQ
DATA
ACK

1601

Fig. 17a

Fig. 17a

Fig. 17

Fig. 18

1805

1804

1801

1806

1802

1803

1801

2201b	compute
2202b	compute pointer A
2207b	load pointer A
2203b	compute
2204b	compute
2205b	compute pointer D
2206b	compute
2210b	compute pointer B with A
2211b	load pointer B
2208b	compute with A
2209b	compute with A
2212b	compute pointer C with B
2213b	compute pointer C
2214b	compute pointer C
2215b	load pointer C

2220b

2221b

2201a	compute
2202a	compute pointer A
2203a	compute
2204a	compute
2205a	compute pointer D
2206a	compute
2207a	load pointer A
2208a	compute with A
2209a	compute with A
2210a	compute pointer B with A
2211a	load pointer B
2212a	compute pointer C with B
2213a	compute pointer C
2214a	compute pointer C
2215a	load pointer C

2220a

2221a

Fig. 22

Fig. 19b

Fig. 19a

Fig. 20b

Fig. 20a

Fig. 21b

Fig. 21a
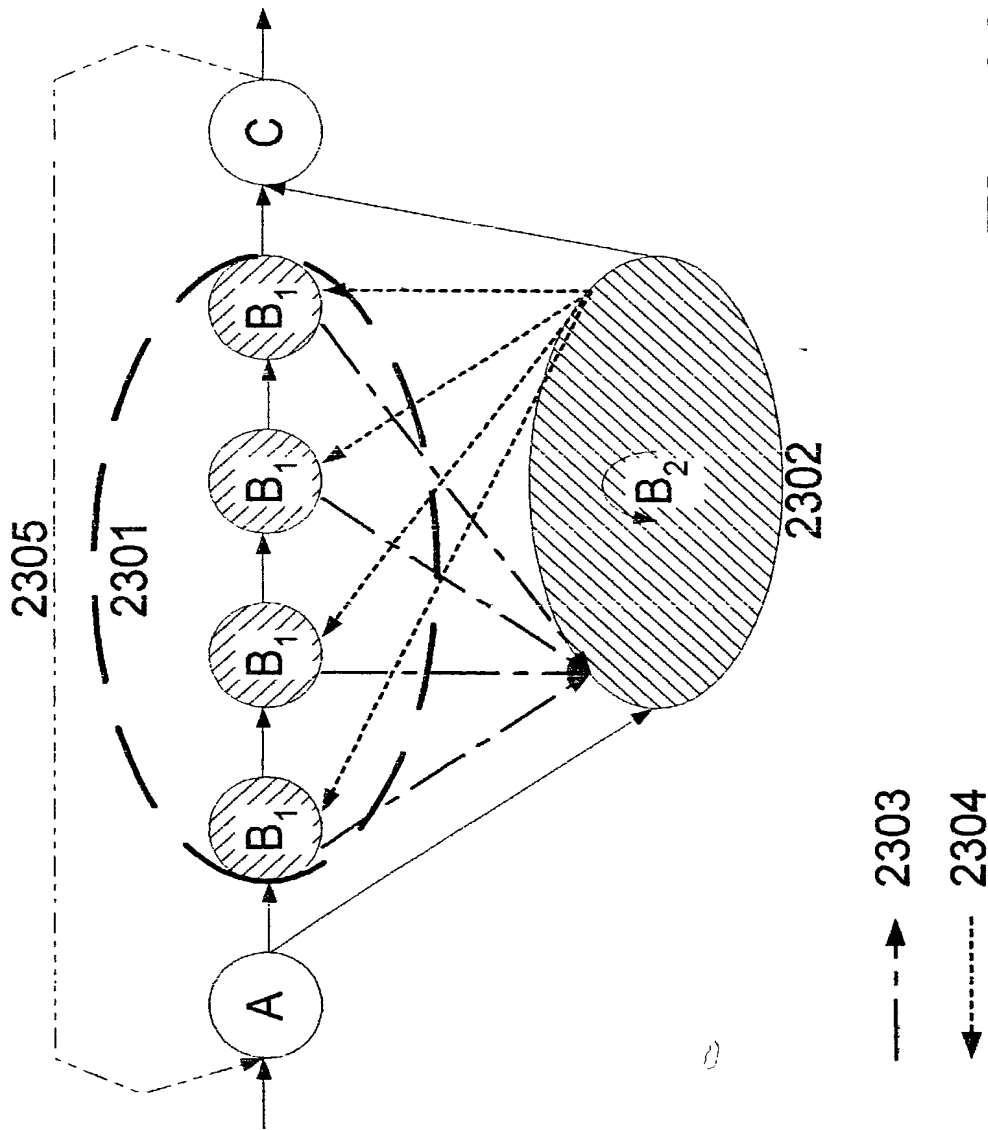
Fig. 23

Fig. 24a

2401

2402

2403

t t+1 t+2 t+3 t+n t+n+1

Fig. 24b

t t+a t+a+b t+a+b+c t+a+b+c+d

Fig. 25

Fig. 26

Fig. 27a
(2602)

Fig. 27
(2602)

Fig. 28

Fig. 29

Fig. 30

Fig. 31c

3102

3105

2912

3104

3101

Fig. 31d

3102

3103

3105

2912

3006

3104

3101

Fig. 31a

3102

2912

3104

3101

Fig. 31b

3102

3103

2912

3006

3104

3101
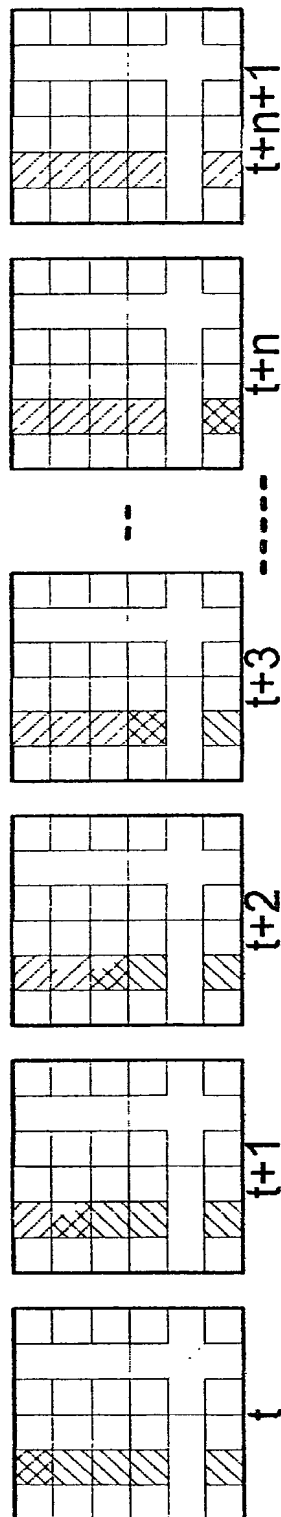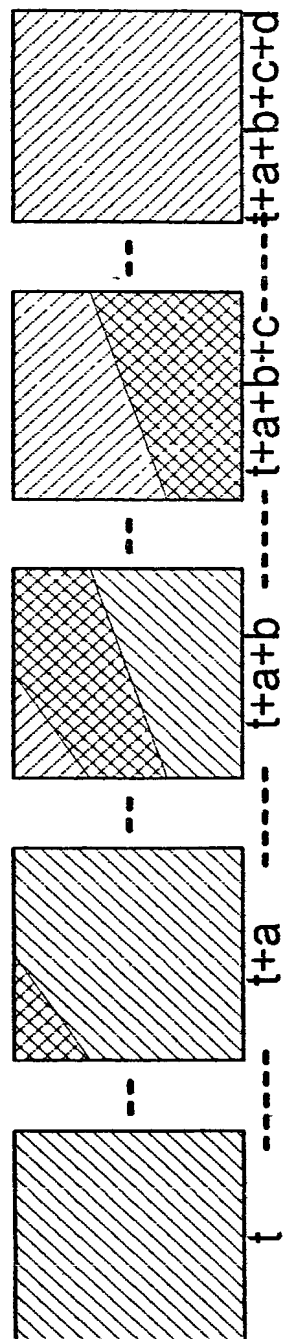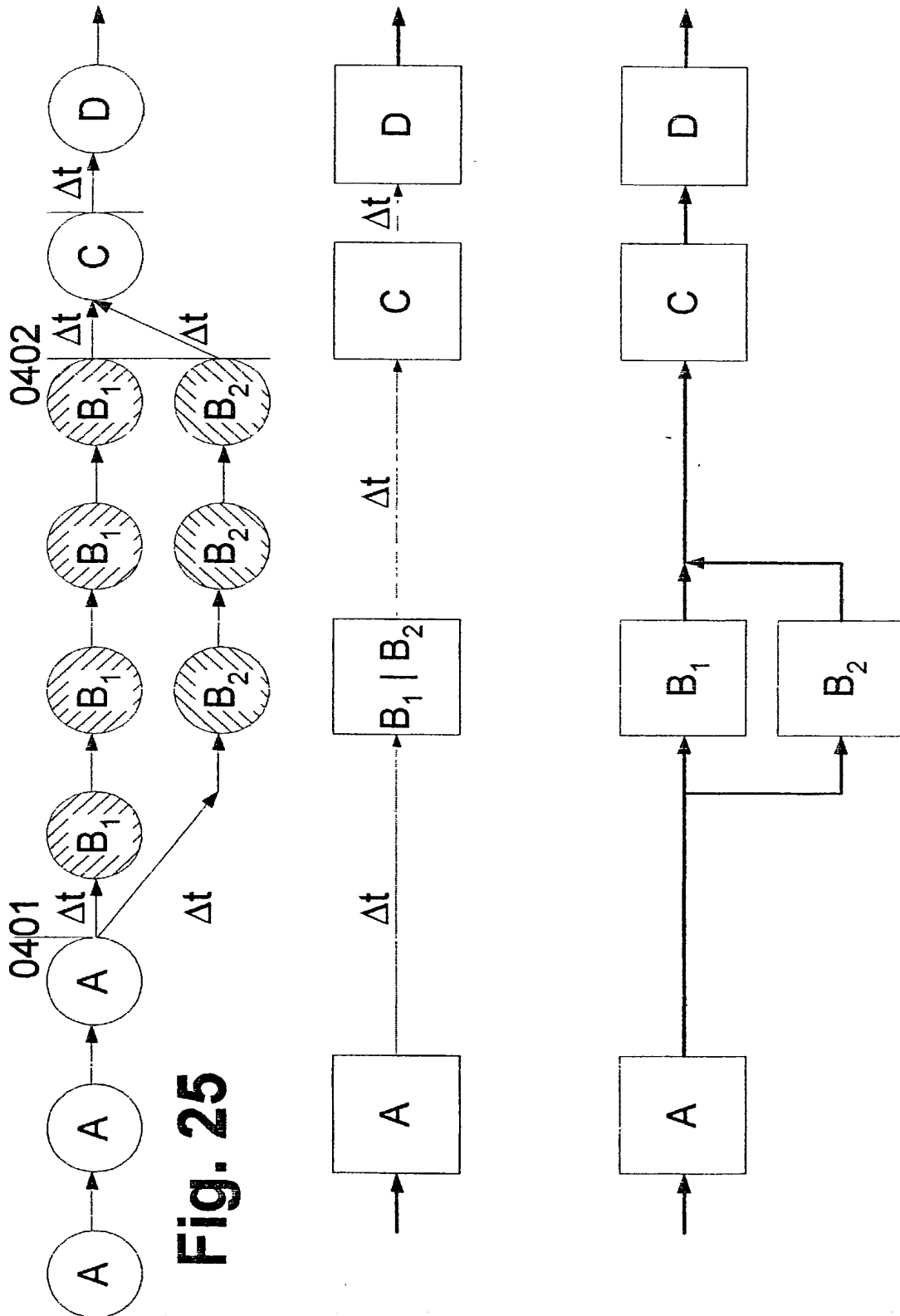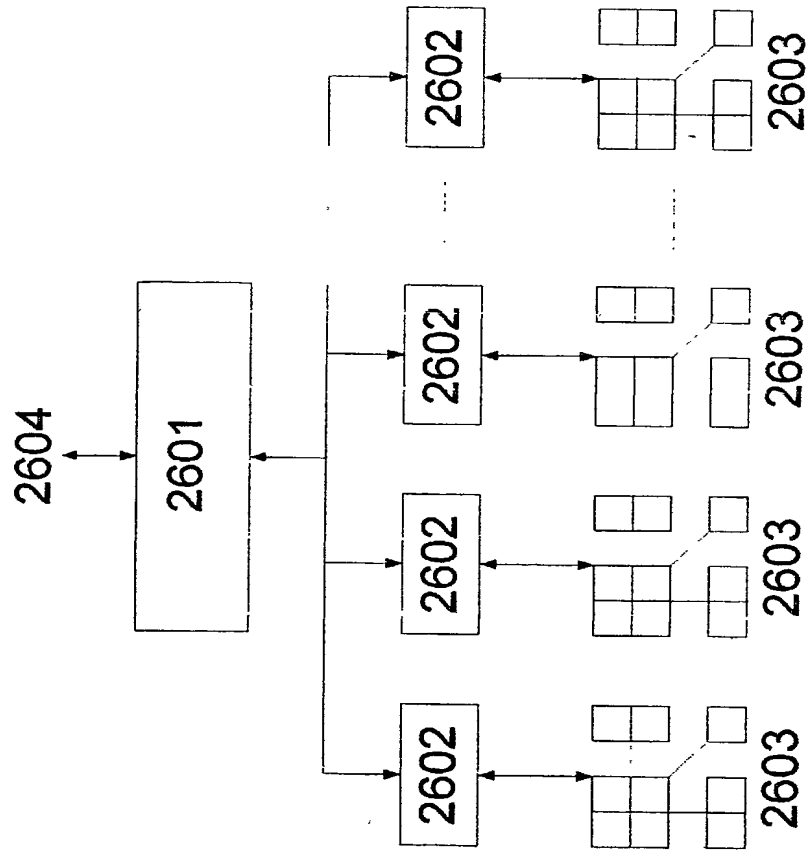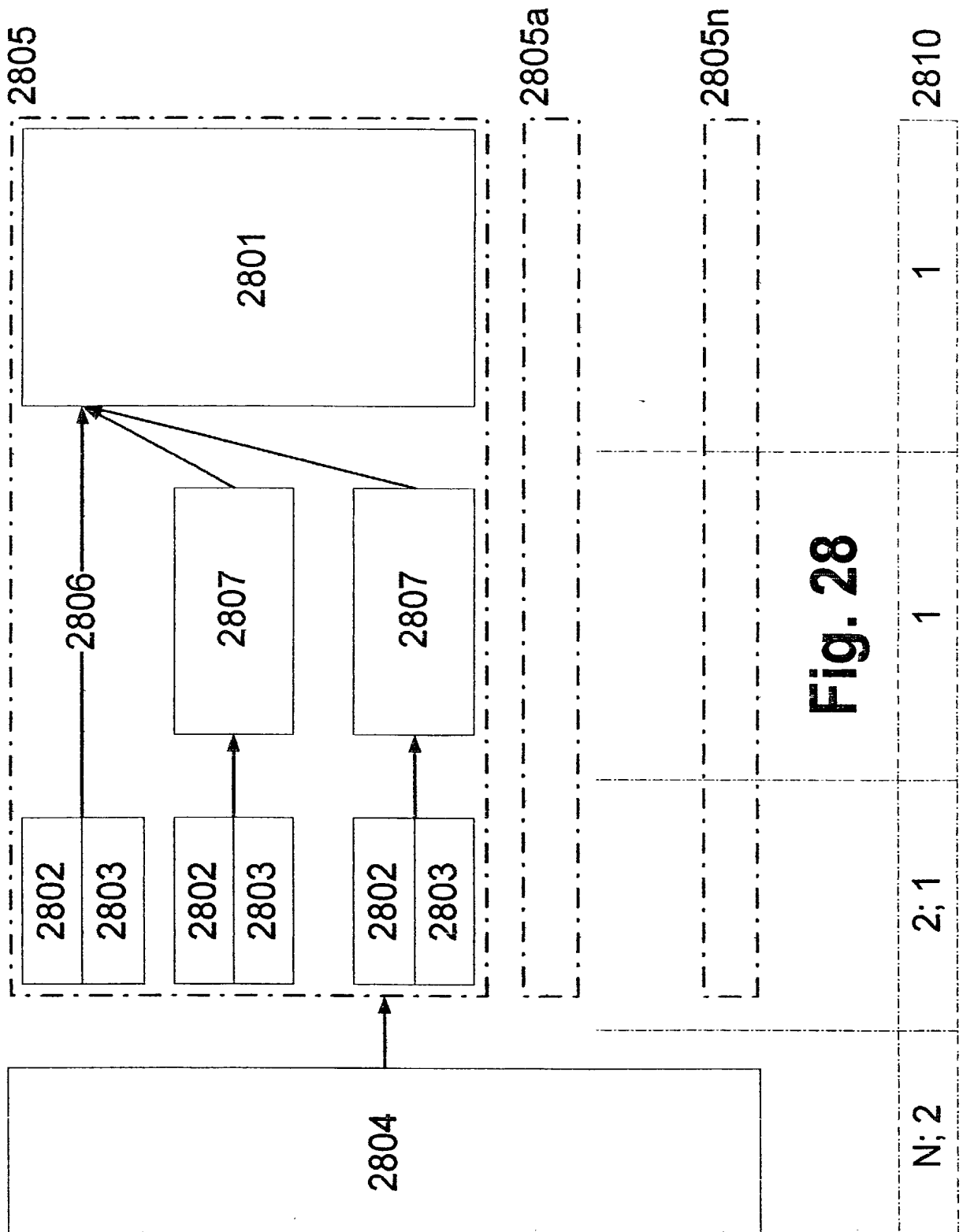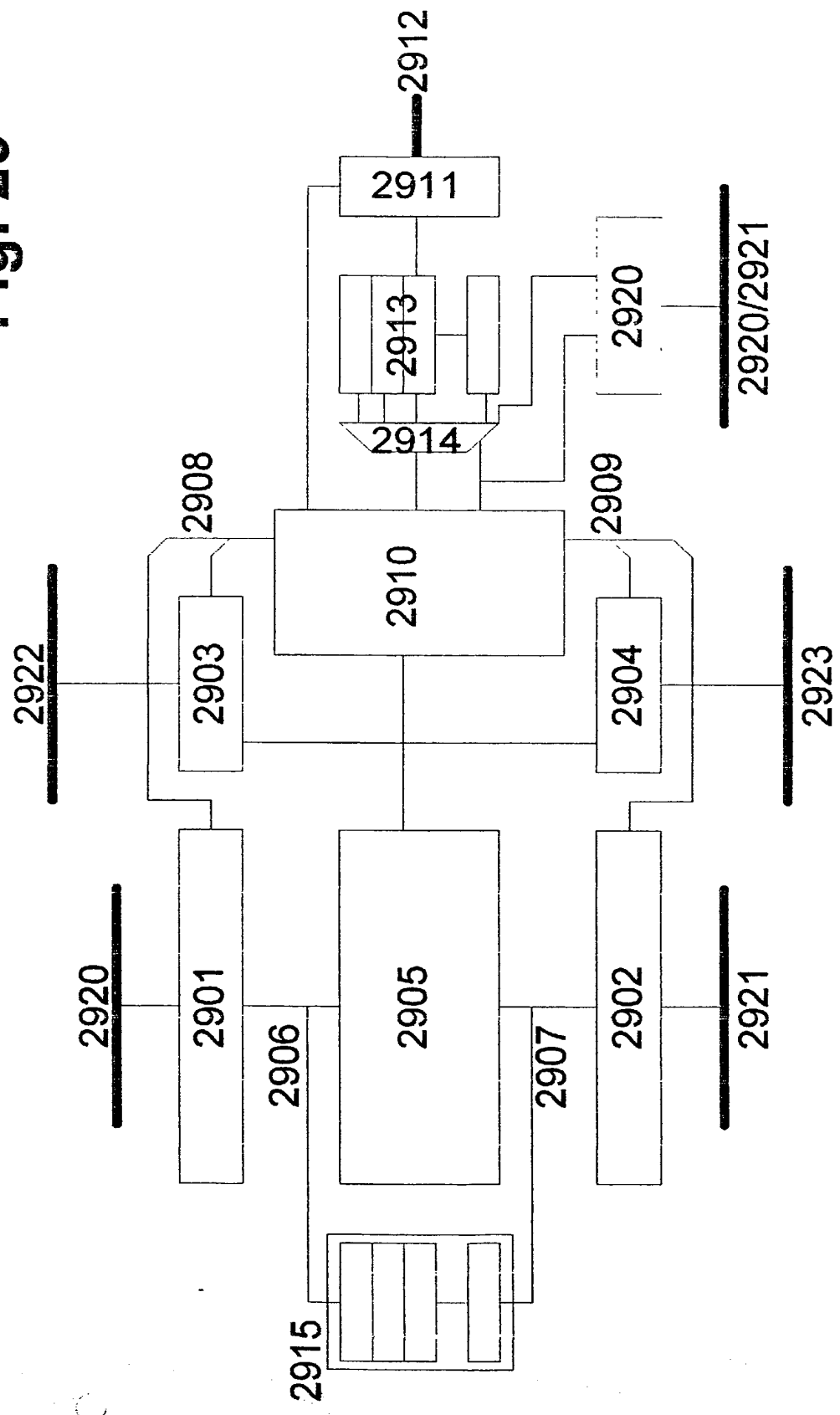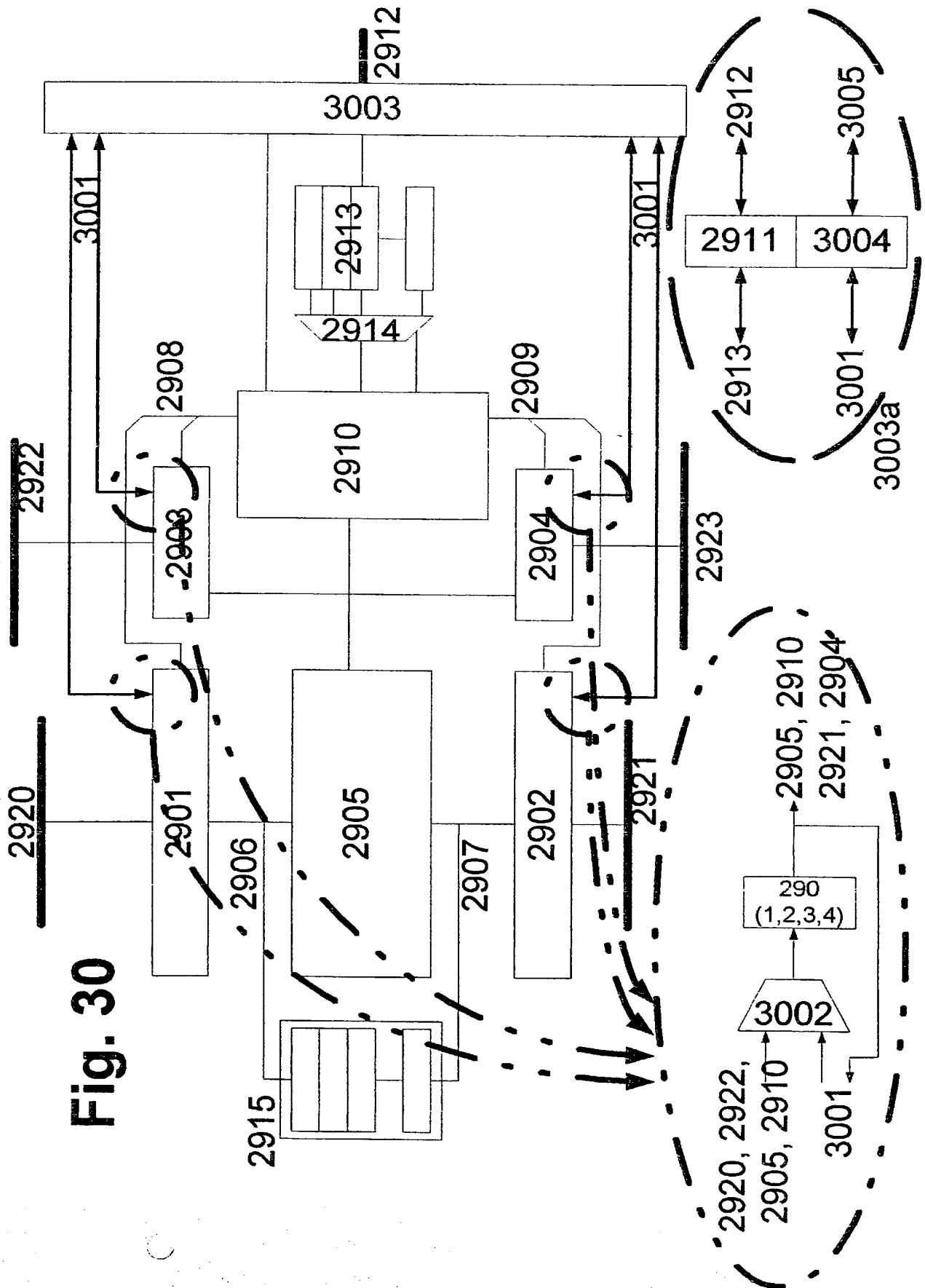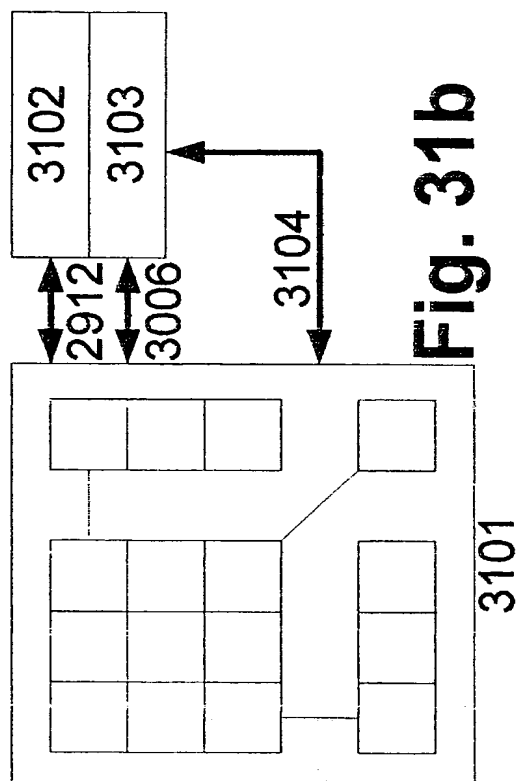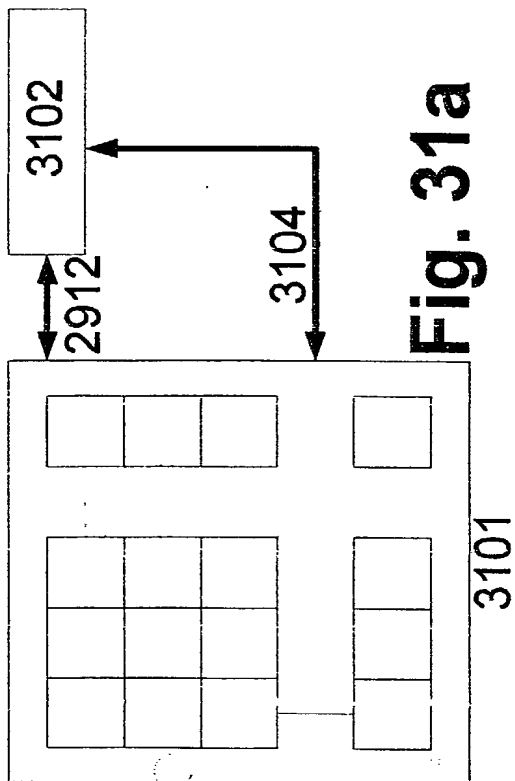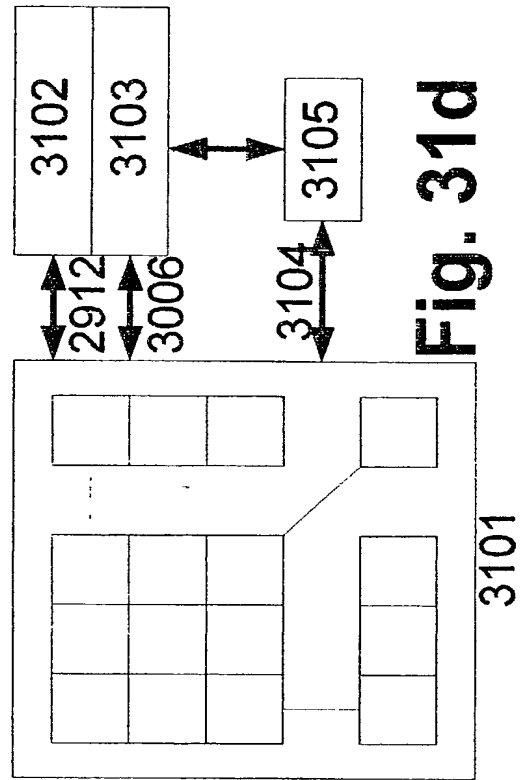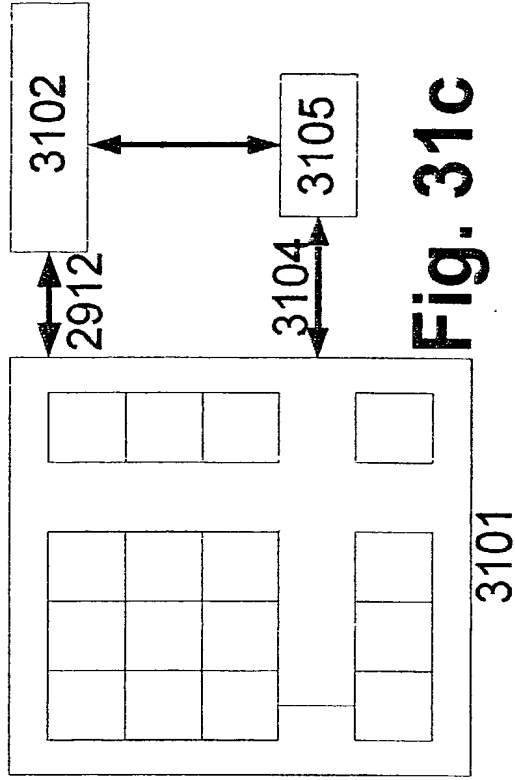
Fig. 32
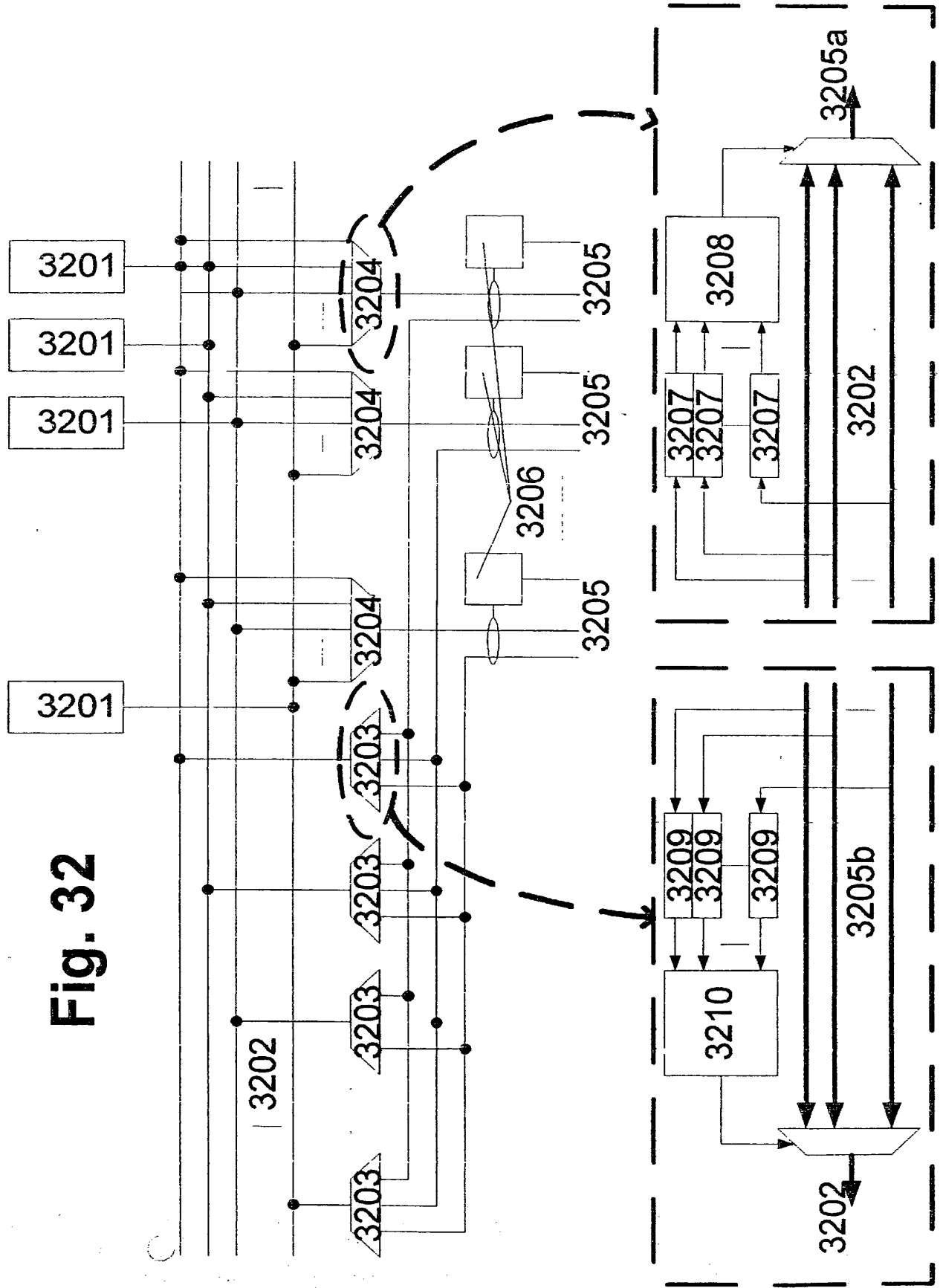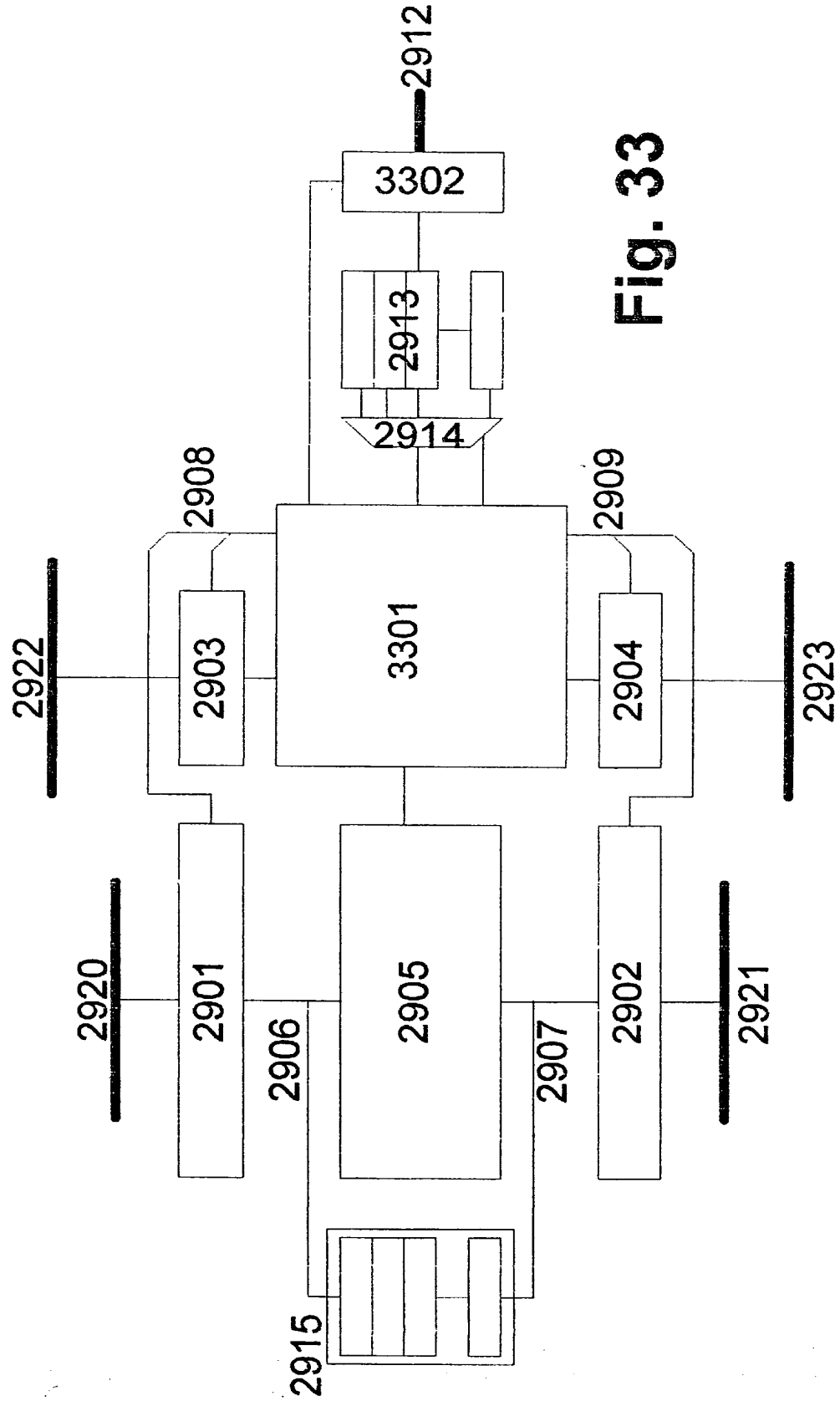
Fig. 33
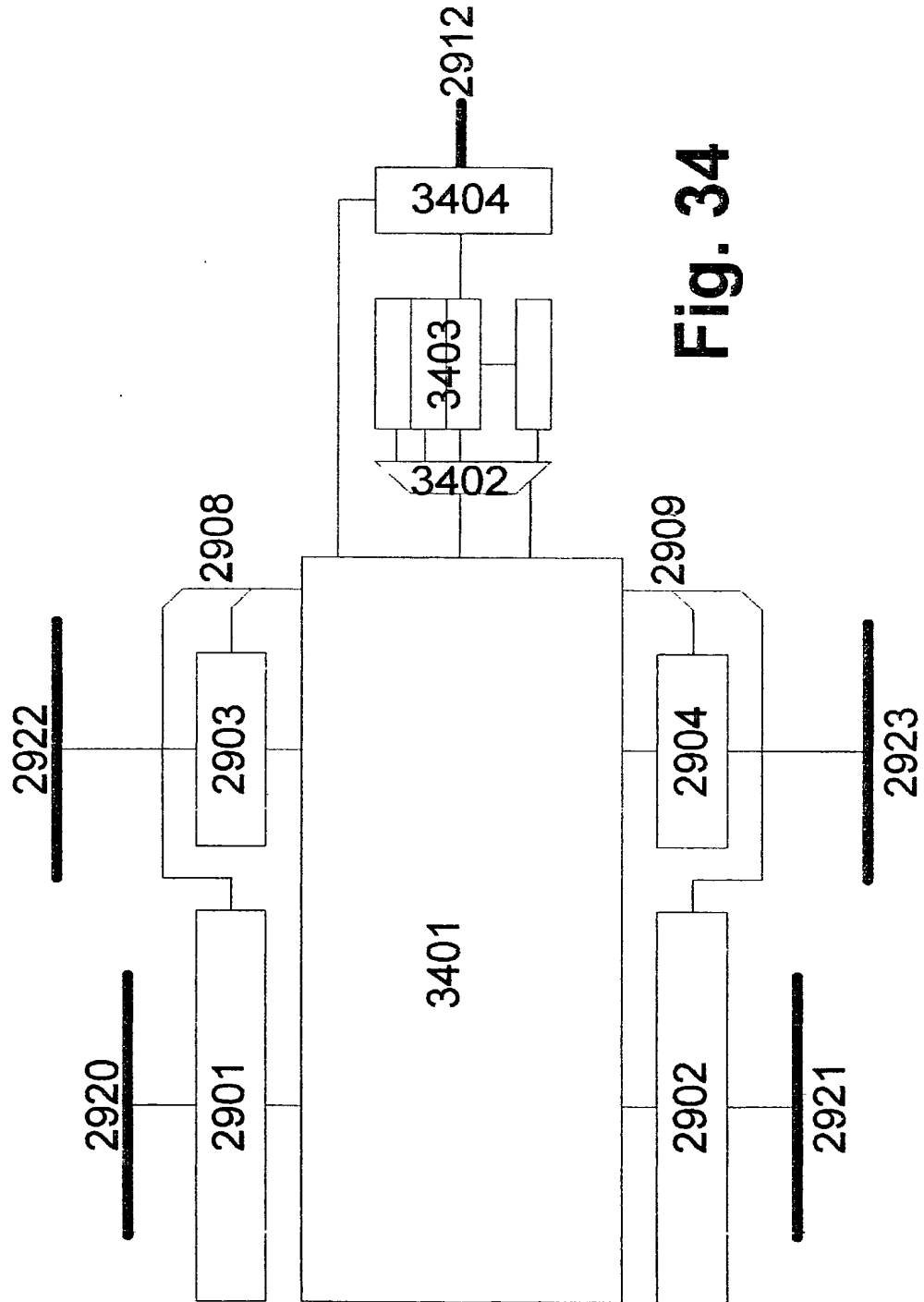
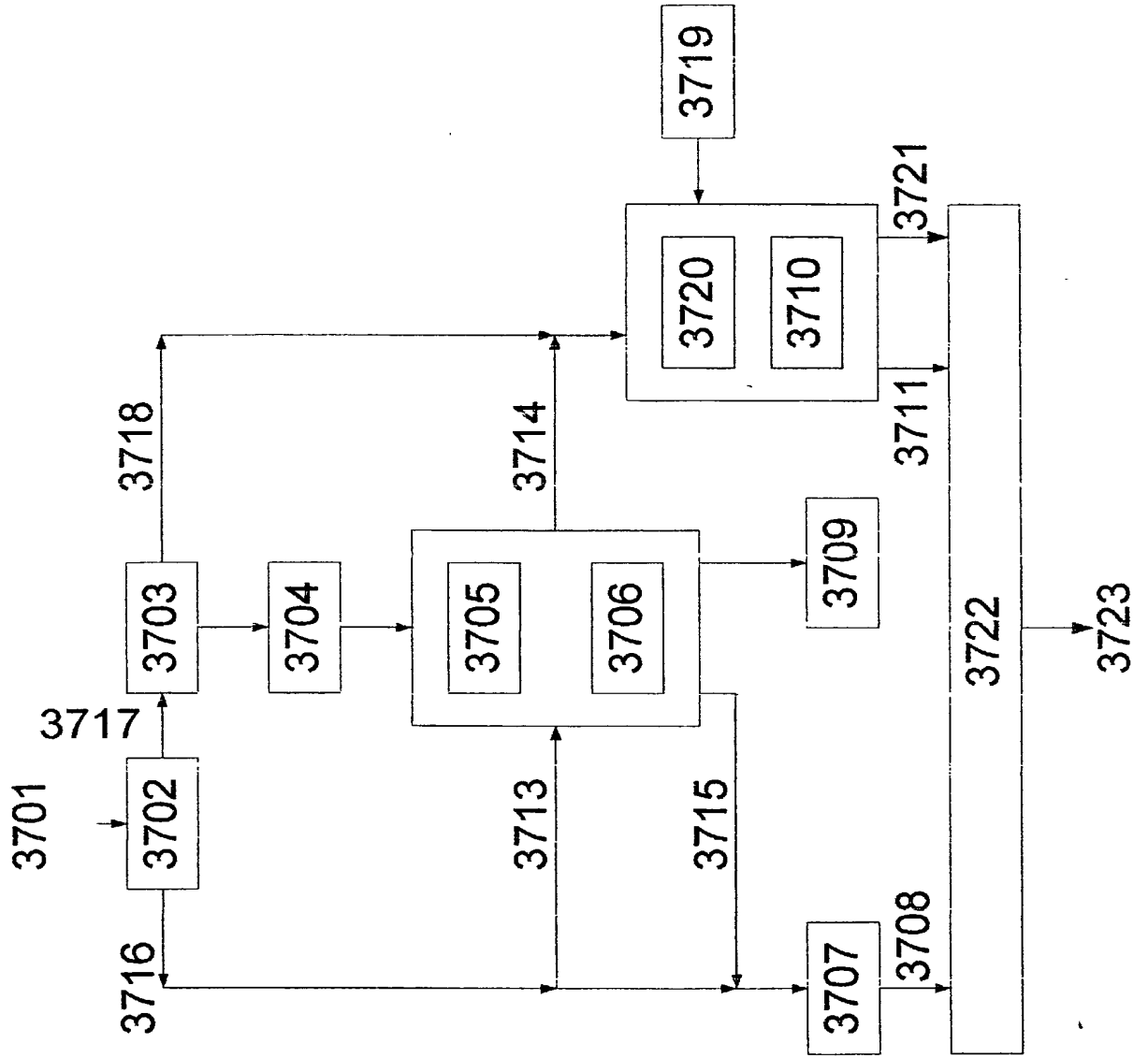Fig. 34
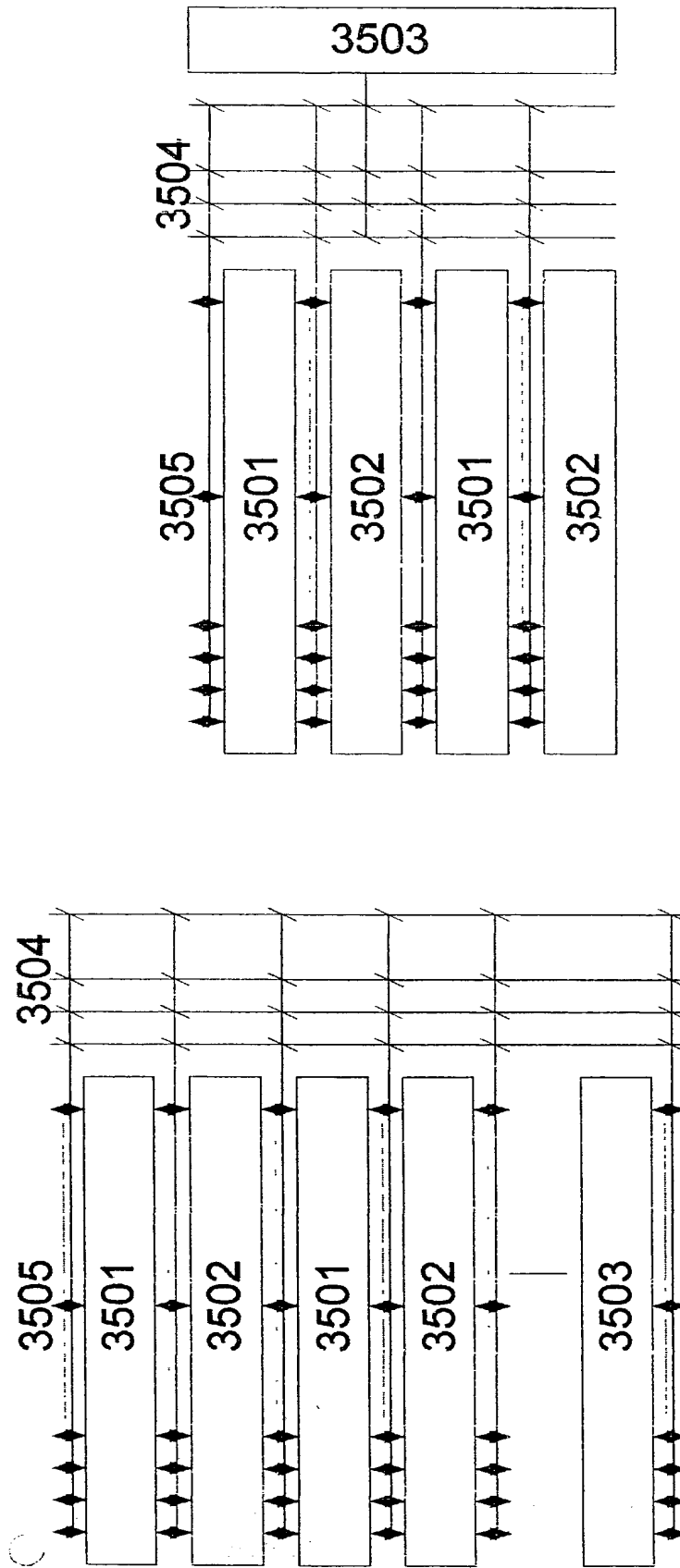
Fig. 35

Fig. 36

Fig. 37

Fig. 38a

Fig. 38
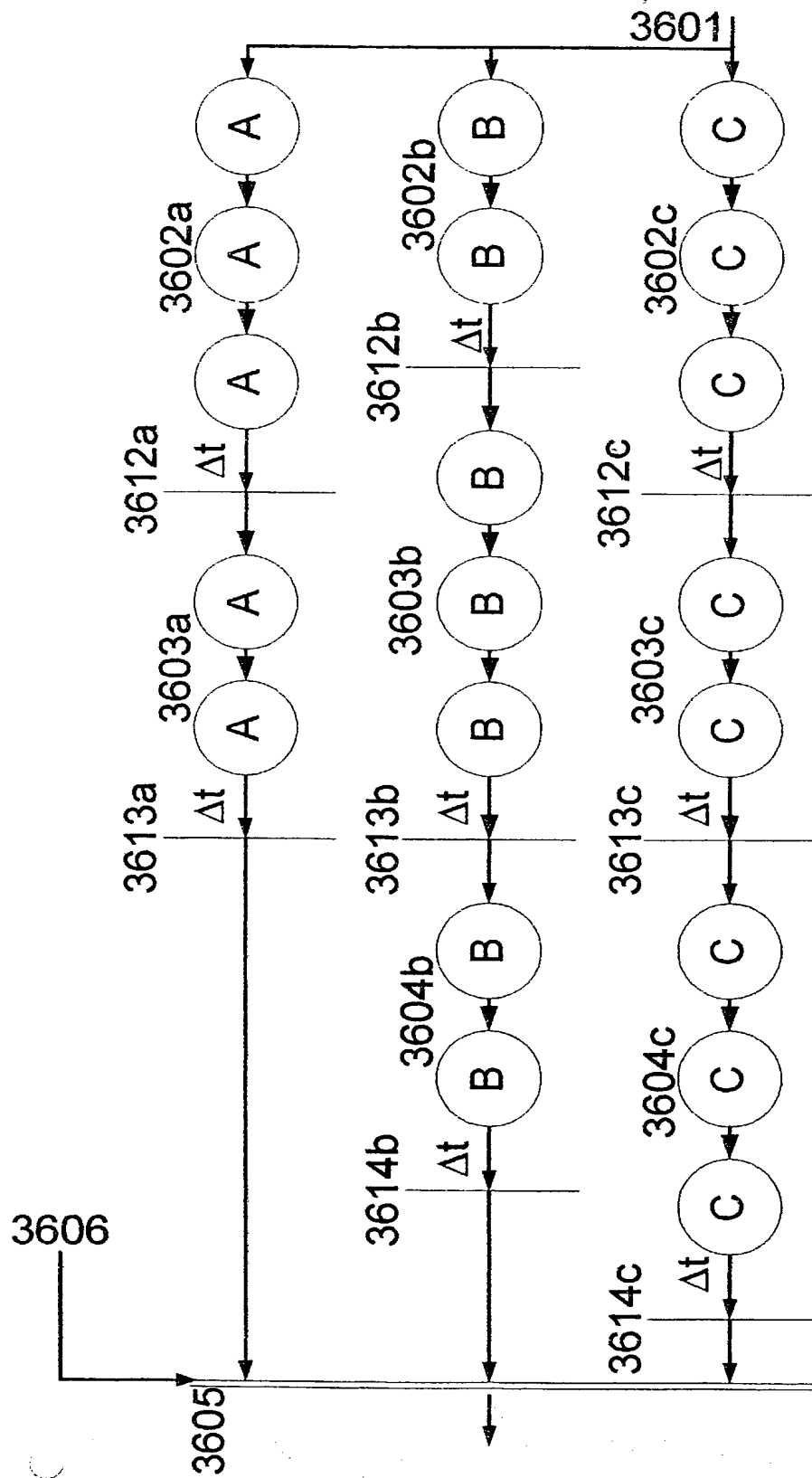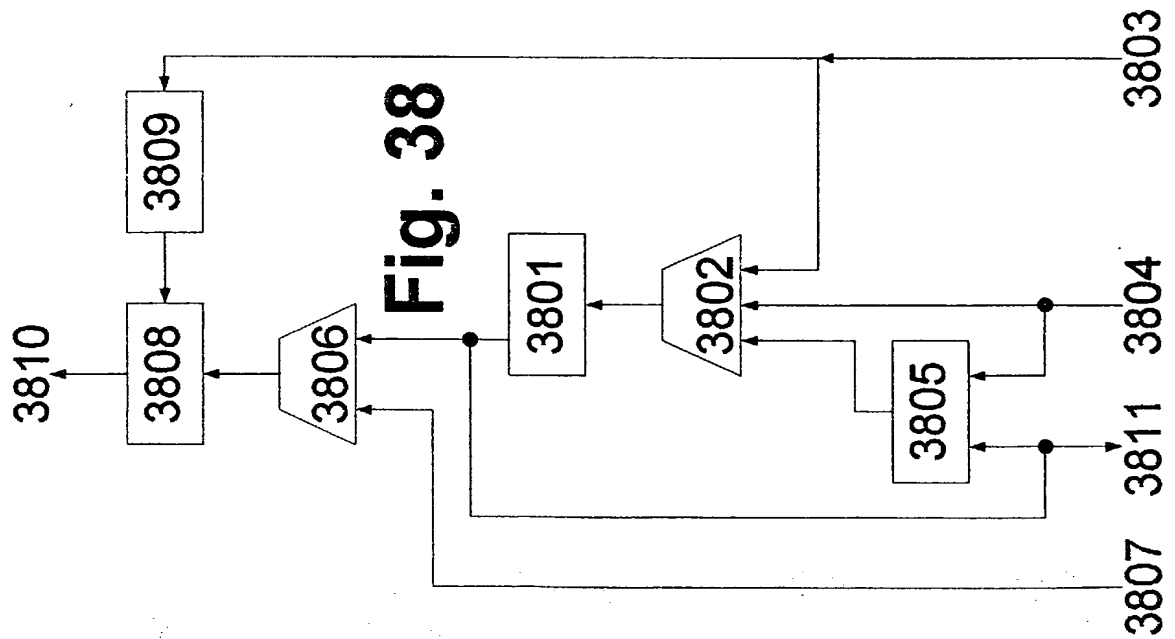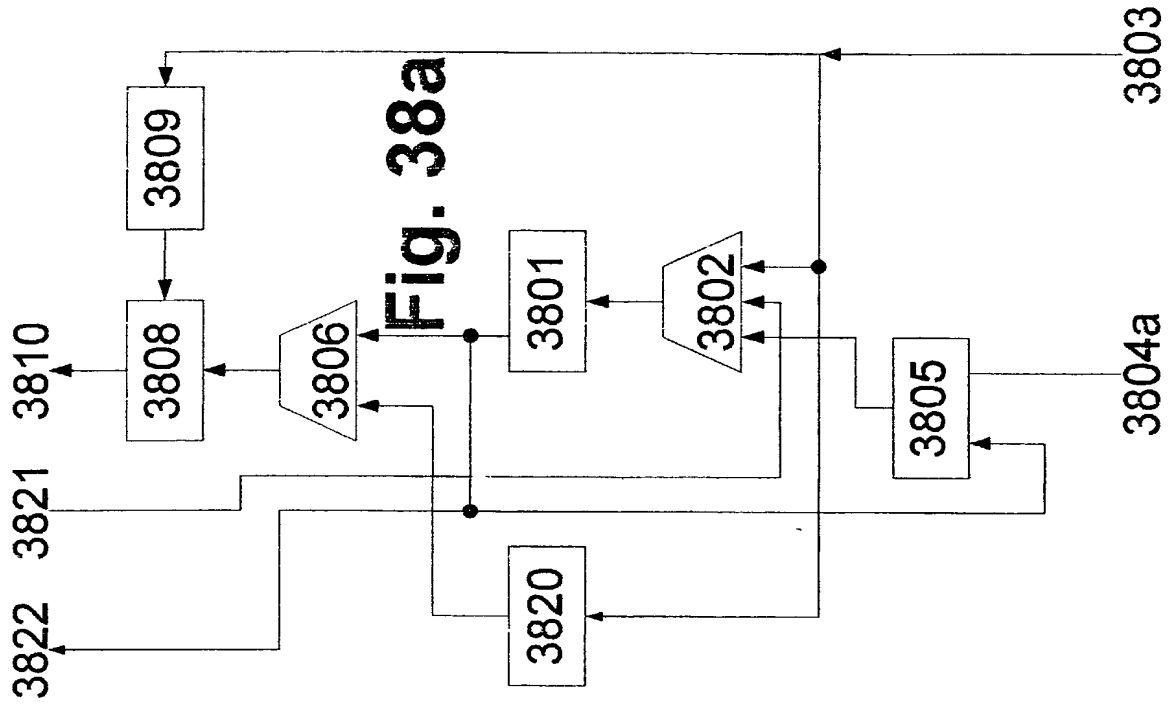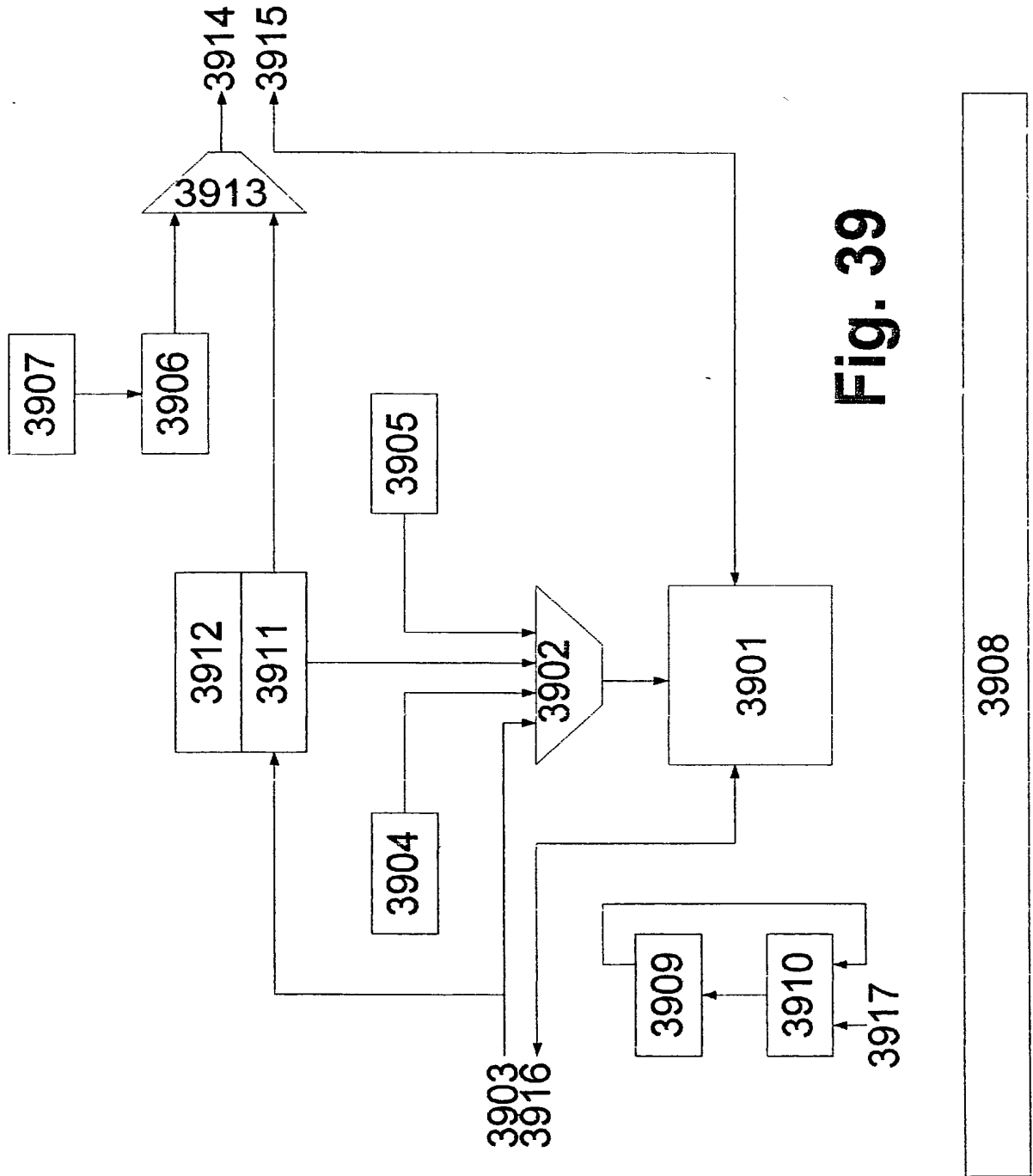
Fig. 39

U.S. DEPARTMENT OF COMMERCE
PATENT AND TRADEMARK OFFICE

DECLARATION	ATTORNEY'S DOCKET NO **2885/56**

As a below named inventor, I hereby declare that:

    My residence, post office address, and citizenship are as stated below next to my name

    I believe I am an original, first, and joint inventor of the subject matter that is claimed and for which a patent is sought on the invention entitled **PROGRAMMING CONCEPTS**, for which an application for Letters Patent was filed as an International Application PCT/DE00/01869 on 13 June 2000.

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims.

I acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, Code of Federal Regulations, § 1.56(a).

PRIOR UNITED STATES APPLICATION(S)

I hereby claim the benefit under Title 35, United States Code, § 119(e) of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, § 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, § 1.56(a) which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION NUMBER	FILING DATE (day, month, year)	STATUS (i.e. Patented, Pending, Abandoned)

PRIOR FOREIGN APPLICATION(S)

I hereby claim foreign priority benefits under Title 35, United States Code, §119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

APPLICATION NUMBER	FILING DATE (day, month, year)	COUNTRY	PRIORITY CLAIMED
DE 199 26 538.0	10 June 1999	Germany	Yes
DE 100 00 423.7	9 January 2000	Germany	Yes
DE 100 18 119.8	12 April 2000	Germany	Yes

SEND CORRESPONDENCE, AND DIRECT TELEPHONE CALLS TO:

Michelle M. Carniaux, Esq.
KENYON & KENYON
One Broadway
New York, New York 10004
(212) 425-7200 (phone)
(212) 425-5288 (facsimile)

I declare that all statements made herein of my own knowledge are true and all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under § 1001 of Title 18 of the United States Code and that such willful statements may jeopardize the validity of the application or any patent issuing thereon.

FULL NAME OF INVENTOR	FAMILY NAME **VORBACH**	FIRST GIVEN NAME **Martin**	SECOND GIVEN NAME
RESIDENCE & CITIZENSHIP	CITY **80689 MUNICH**	STATE OR FOREIGN COUNTRY **GERMANY**	COUNTRY OF CITIZENSHIP **GERMANY**
POST OFFICE ADDRESS	POST OFFICE ADDRESS **GOTTHARDSTRASSE 117A**	CITY **80689 MUNICH**	STATE & ZIP CODE/COUNTRY **GERMANY**
Signature		Date 23rd may 2002	

FULL NAME OF INVENTOR	FAMILY NAME **NÜCKEL**	FIRST GIVEN NAME **Armin**	SECOND GIVEN NAME
RESIDENCE & CITIZENSHIP	CITY **D-76777 NEUPOTZ**	STATE OR FOREIGN COUNTRY **GERMANY**	COUNTRY OF CITIZENSHIP **GERMANY**
POST OFFICE ADDRESS	POST OFFICE ADDRESS **DROSSELWEG 4**	CITY **D-76777 NEUPOTZ**	STATE & ZIP CODE/COUNTRY **GERMANY**
Signature		Date 23rd may 2002	

NY01 429902 v 1

APPOINTMENT OF POWER OF ATTORNEY
BY ASSIGNEE OF ENTIRE INTEREST

PACT INFORMATIONSTECHNOLOGIE GmbH as assignee of the entire right, title,

and interest in the application for patent entitled **PROGRAMMING CONCEPTS**, for which an

application for Letters Patent was filed as an International Application Application

PCT/DE00/01869 on 13 June 2000, does hereby appoint Edward J. Handler, III (Registration

No. 25,597), Michelle M. Carniaux (Registration No. 36,098) and Andrew L. Reibman

(Registration No. 47,893), as my attorneys with full power of substitution and revocation, to

prosecute this application and to transact all business in the Patent and Trademark Office

connected therewith.

Please address all communications regarding this application to:

Michelle M. Carniaux, Esq.
KENYON & KENYON
One Broadway
New York, New York 10004

26646
PATENT TRADEMARK OFFICE

Please direct all telephone calls to Michelle M. Carniaux at (212) 425-7200.

PACT INFORMATIONSTECHNOLOGIES GmbH
Leopoldstr. 236
80807 München
FEDERAL REPUBLIC OF GERMANY

Dated: 23rd may 02

By:_____
Name :   Martin Vorbach
Position: CTO
PACT INFORMATIONSTECHNOLOGIE GmbH

429907